Managing Schema Change in an Heterogeneous Environment

by

Kajal Tilak Claypool

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

May 2002

APPROVED:

Prof. Elke A. Rundensteiner
Advisor

Prof. Carolina Ruiz
Committee Member

Prof. George T. Heineman
Committee Member

Prof. Stan Zdonik
External Committee Member
Brown University, Providence, RI

Dr. Arnon Rosenthal
External Committee Member
MITRE Corporation, Burlington, MA

Prof. Micha Hofri
Head of Department

Abstract

"Nothing endures but change. "

–By Heractilus

Change is inevitable even for persistent information. Effectively managing change of persistent information, which includes the specification, execution and the maintenance of any derived information, is critical and must be addressed by all database systems. Today, for every data model there exists a well-defined set of change primitives that can alter both the structure (the schema) and the data. Several proposals also exist for incrementally propagating a primitive change to any derived information (or view). However, existing support is lacking in two ways. First, change primitives as presented in literature are very limiting in terms of their capabilities allowing users to simply add or remove schema elements. More complex types of changes such the merging or splitting of schema elements are not supported in a principled manner. Second, algorithms for maintaining derived information often do not account for the potential heterogeneity between the source and the target. The goal of this dissertation is to provide solutions that address these two key issues.

The first part of this dissertation addresses the challenge of expressing

a rich complex set of changes. We propose the SERF (Schema Evolution through an Extensible, Re-usable and Flexible) framework that allows users to perform a wide range of complex user-defined schema transformations. Our approach combines existing schema evolution primitives using OQL (object query language) as the glue logic. Within the context of this work, we look at the different domains in which SERF can be applied, including web site management. To further enrich our framework, we also investigate the optimization and verification of SERF transformations.

The second part of this dissertation addresses the problem of maintaining views in the face of source changes when the source and the view are not in the same data model. With today's increasing heterogeneity in information structure, it is critical that maintenance of views addresses the data model boundaries. However, view definitions that go across data models are limited to hard-coded algorithms, thereby making it difficult to develop general maintenance algorithms. We provide a two-step solution for this problem. We have developed a cross algebra, that defines views such that there is no restriction that forces the view and the source data models to be the same. We then define update propagation algorithms that can propagate changes from source to target irrespective of the exact translation and the data models. We validate our ideas by applying them to translation and change propagation between the XML and relational data models.

Acknowledgments

There are many people who have helped me in this work, either directly by working on projects with me or indirectly by giving me vision and support. However, above all there are three people who deserve my deepest gratitude: Professor Elke Rundensteiner, my husband, Mark and my Papa. As my advisor, Elke has contributed to all levels of this thesis, from guidance on English grammar rules to providing focus and vision for my research.

Equally important, Mark has been a pillar of strength and support. He has been understanding of the many late nights and the many weekends I spent away from him and our two kids. Most importantly, he has provided motivation to persevere through the slow and difficult parts of my dissertation, and more importantly, helped me keep my focus on finishing when I may have given up. He is my inspiration in more ways than one.

My father taught me how to set my sights high and how to succeed. His motto "just work hard" did well for me during my years at school. In

many ways he is the reason that I did this. So Papa, this one is for you!

I would also like to thank my mother, my brother, my two boys and my in-laws. Akaash and Saahil, thank you for the infinite patience you showed me when I was gone on the weekends and when I had to do my homework before I could help you with yours! Thank you boys. Thank you Mom for being supportive and encouraging and helping out with the kids whenever you could. Thanks Amit for patiently listening to me whine! Last but not the least in the family line - thanks to my parents-in-law for being very supportive.

I would like to thank the members of my examining committee: Professors George Heineman, Carolina Ruiz, Stan Zdonik and Dr. Arnie Rosenthal. They helped steer the direction of my dissertation and gave me valuable advice along the way. Professor George Heineman provided hands-on advising when I needed it most.

I also must thank those who long ago gave me the tools to achieve a Ph.D. Professors John Riedl and Jaideep Srivastava, professors at University of Minnesota, showed me my first glimmer of research and had me hooked.

I would like to thank my colleagues and friends in the DSRG group, especially Li Chen, Andreas Koeller and Su Hong, for their invaluable encouragement and contributions to my work, and most of all for lending me an ear when I needed it the most. Thank you.

Without the contributions of the above people, this work would never have been completed, nor even begun.

Contents

I Information Integration and Change Management 1

1 Introduction 3
- 1.1 Issues in Change Management 3
- 1.2 Change Management - State of Art 5
  - 1.2.1 Change Specification and Execution 5
  - 1.2.2 Managing the Effects of Change 10
- 1.3 Our Work 15
  - 1.3.1 Change Specification - The SERF Framework 16
  - 1.3.2 Managing the Effects of Change - Sangam 20
- 1.4 Organization of this Dissertation 27

II SERF - An Extensible Transformation Framework 29

2 Overview 31

3 The ODMG Model and Schema Evolution 33
- 3.1 ODMG Standard: The Object Model 33
  - 3.1.1 ODMG Schema Repository 35
  - 3.1.2 ODMG's Object Query Language - OQL 37
- 3.2 Evolving the ODMG Object Model 39
  - 3.2.1 Invariants for the ODMG Object Model 40
  - 3.2.2 Taxonomy of Schema Evolution Primitives 42
- 3.3 Summary 50

4 The SERF Framework 51
- 4.1 Features of the Framework 51
- 4.2 Schema Transformations 54
- 4.3 Transformation Templates 56
  - 4.3.1 Generalization of a Transformation 57

4.3.2 Template Library 58
4.3.3 SERF Template Language 59
4.3.4 SERF Template Instantiation and Processing 59
4.4 Summary 61

5 Soundness and Consistency of SERF Templates 63
5.1 Soundness of Templates 63
5.2 Contracts 66
5.3 Verification of ROVER Wrappers 69
5.3.1 Theorem Prover for SERF Templates 71
5.3.2 Formal Verification Process: Application to Inline Template 73
5.4 Summary 80

6 Reducing the Runtime of SERF Templates 83
6.1 Optimizing the Performance of Schema Evolution Sequences 83
6.1.1 Foundations of Schema Evolution Sequence Analysis 85
6.1.2 The CHOP Optimization Functions 89
6.1.3 CHOP Optimization Strategy 96
6.1.4 Experimental Validation 98
6.2 Summary 100

7 Design and Implementation of the OQL-SERF System 103
7.1 System Architecture 104
7.2 SERF Framework Modules 105
7.2.1 Template Module 105
7.2.2 Template Library 107
7.2.3 User Interface 108
7.3 Summary 109

8 Case Study of Different Schema Transformations 111
8.1 The Merge Transformation 112
8.1.1 The Merge-Union Template 113
8.1.2 The Merge-Difference Template 114
8.1.3 A More Complex Merge Template 115
8.2 ReWEB - Applying SERF for Web Transformations 117
8.3 Summary 122

9 Related Work 123
9.1 Specifying Schema Change 123
9.2 Optimization of Schema Evolution 127
9.3 Behavioral Consistency 128
9.4 Consistency Management 128
9.5 Extensible Systems 129

III Sangam - Managing Change in Heterogeneous Databases 131

10 Overview 133

11 Data Model 137
11.1 Sangam Graph Model 138
11.2 Converting Application Schemas and Data to Sangam graphs 141
11.2.1 XML and Sangam graphs 141
11.2.2 Relational Schemas and Sangam graphs 150
11.3 Summary 154

12 Cross Algebra 157
12.1 Cross Algebra Operators 158
12.1.1 The Cross Operator 158
12.1.2 The Connect Operator 160
12.1.3 The Smooth Operator 161
12.1.4 The Subdivide Operator 162
12.1.5 Notation 165
12.2 Cross Algebra Trees 165
12.2.1 Derivation Trees 167
12.2.2 Context Dependency Trees 173
12.2.3 Cross Algebra Graphs (CAG): Combining Context Dependency and Derivation Trees 179
12.3 Evaluating the Cross Algebra Operators - The Physical Plan 193
12.3.1 Physical Algebra Operators 193
12.3.2 Evaluating a Cross Algebra Graph 197
12.4 An Example 205
12.4.1 The Basic Inlining 205
12.4.2 The Shared Inlining 214
12.5 Summary 216

13 Sangam: Evaluating the Theory 219
13.1 Architectural Overview of Cross Algebra Processing Engine 219
13.2 Experimental Validation of Sangam 222
13.2.1 Loading to and Generating from Sangam Graphs . . 224
13.2.2 CAG Evaluation 228
13.2.3 Summary of Experimental Results 234
13.3 Summary 235

14 Updating the Sangam Graph 237
14.1 Schema Change Operations on the Sangam Graph 238
14.1.1 SAG-SC: Schema Change Primitives 238
14.1.2 Completeness of SAG-SC Operations 240
14.1.3 Correctness of Sangam graph -SC Operations 243
14.2 Data Modification Primitives for the Sangam Graph 244
14.3 Translation Completeness of Sangam Graph Operations . . . 246
14.3.1 Translation Completeness of SAG-SC Operations . . 246
14.3.2 Translation Completeness of SAG-DU Operations . . 251
14.4 Reversing the Process 259
14.5 Summary 260

15 Update Propagation 263
15.1 The CAT Propagation Strategy 264
15.1.1 Introduction 264
15.1.2 Updates on CAT 264
15.1.3 Overall Propagation Strategy 265
15.2 Default Propagation Through the Cross Algebra Operators . 281
15.2.1 Propagation Rules for SAG-SC Operations 281
15.2.2 Propagation Rules for SAG-DU Operations 293
15.3 Properties of Propagation Through CAGs 297
15.3.1 Correctness of Propagation 297
15.3.2 Incremental Propagation Versus Recomputation . . . 300
15.4 Summary 306

16 Performance Comparison: Incremental vs Re-computation 309
16.1 Incremental vs. Re-Computation 312
16.2 Analyzing the Performance of Incremental Propagation . . . 313
16.2.1 The Algebra Operators 314
16.2.2 Ident and Inline CAGs 323
16.3 Summary 328

17 Related Work **331**
17.1 XML Storage 331
17.2 Change Management 332
17.2.1 Change Propagation within A Data Model 332
17.2.2 Supporting Applications During Change 334
17.2.3 XML Update Systems 336
17.2.4 Other Related Work. 337
17.3 Schema Integration and Data Transformation 337
17.4 Data Models and Translations in the Middle Layer 339
17.4.1 Schema Languages 339
17.4.2 Query Algebras 348

IV Conclusions and Future Work **361**

18 Conclusions and Future Work **363**
18.1 Contributions of this Dissertation 364
18.1.1 Support for Complex Changes 364
18.1.2 Maintenance of Heterogeneous Views 366
18.2 Future Directions 368
18.2.1 Optimization of Schema Evolution 368
18.2.2 Integrating the Cross Algebra With Existing Local Algebra 370
18.2.3 Inverse Update Propagation 371

A DTDs Used in Experiments **373**

List of Figures

1.1 Changes in the Database Environment. 4
1.2 The Inline Operation. 18

4.1 An Example Schema Graph for a MERGE Transformation. . 53
4.2 Example of the Inline Transformation. 54
4.3 SERF Representation of the Inline Transformation using OQL. 54
4.4 Generalized Inline Template based on the Inline Transformation. 58
4.5 The BNF for a SERF Template. 60
4.6 Steps for the Execution of a Template 61

5.1 Preconditions for Delete-Class Primitive in Contractual Form. 68
5.2 Postconditions for the Delete-Class Primitive Template. We assume here that the meta-dictionary information such as $P_e(\mathtt{C}_x)$ are available until the template is done executing. . . 68
5.3 Inline ROVER Wrapper with Set-Theoretic Contracts. 69
5.4 Add-Attribute Primitive Template with Contracts. 74

6.1 Best and Worst Case Sequence Times w/o Algorithm Overhead for Input Sequences of Length 8 on the Sample Schema. 99
6.2 Distribution: Number of Classes = Sequence Length. 99

7.1 Architecture of the SERF Framework 104
7.2 The Template Module. 106
7.3 The Template Class. 106
7.4 Steps for the Execution of a Template 107
7.5 The Template Editor. 108
7.6 The Schema Viewer. 108

8.1 Merge-Union: The Structure of the New Class given by Union of the Properties of the Two Source Classes `Authors`, `Papers`. 114

8.2 Merge-Difference: The Structure of the New Class given by Difference of the Properties of the Two Source Classes Authors, Papers. 114
8.3 The Merge-Union Template. 115
8.4 The Merge-Difference Template. 116
8.5 The Merge-Common Template - Shows the Common Code between the Merge-Union and the Merge-Difference templates. 117
8.6 The ODMG Data Model and Corresponding XML Files for the Original Web Site. 119
8.7 The Generated Example Home Pages of Original Web Site . 120
8.8 Nested Inline Template by Re-using Basic Inline Template. . 121
8.9 The ODMG Data Model and Corresponding XML Files for Desired Web Site 121
8.10 The Generated Example Home Pages of Desired Web Site . . 121

10.1 An Example Relational Schema. 134
10.2 A Fragment of the XMark Benchmark Schema [SWK+01]. . . 135
10.3 Modified XMark Benchmark Schema [SWK+01]. A New SubElement reserve is Added. The New Additions are Shown in Bold. 135

11.1 The LoadXML Algorithm to Translate an XML DTD into a Sangam graph. 143
11.2 The LoadXML Algorithm to Translate an XML DTD into a Sangam graph - The Second Pass. 145
11.3 A Fragment of the XMark Benchmark DTD as shown in Figure 10.2 depicted as a Sangam Graph. No order and quantifier annotations are shown for the backpointer edges to distinguish them in the figure. These are defaulted to 1 and [1:1] respectively. 146
11.4 A Fragment of the XMark Benchmark Document Conforming to the XMark Benchmark Schema in Figure 10.2. 149
11.5 The Extent of the Sangam graph in Figure 11.3 based on the XML Document given in Figure 11.4. Part (a) depicts the extent. Here we show the extent of each node, and the extent of edge e:<item, location>. Part (b) presents just the object structure. 150
11.6 An Example Relational Schema. 151

11.7 Relational Schema of Figure 11.6 depicted as a Sangam graph. 152

12.1 (a) The Cross Algebra Operator; (b) The Connect Algebra Operator. 159
12.2 The Functionality of the Smooth Operator. 162
12.3 An Example of the Smooth Operator. 162
12.4 The Functionality of the Subdivide Node. 163
12.5 An Example of the Subdivide Node. 163
12.6 Derivation Composition. 168
12.7 Context Dependency Composition Example. 174
12.8 A Cross Algebra Tree. 181
12.9 Another Example of a Cross Algebra Graph. 188
12.10 The Cross Physical Operator - An Implementation. 198
12.11 The Connect Physical Operator - An Implementation. 199
12.12 The Smooth Physical Operator - An Implementation. 200
12.13 The SubDivide Operator - An Implementation. 201
12.14 The Evaluation Algorithm for a Cross Algebra Graph. 202
12.15 Fragment of Relational Schema Generated by Mapping the XMark Benchmark Schema of Figure 10.2 using the Basic Inlining Technique [STZ+99]. All field sizes are set during the mapping of atomic nodes which are not shown here. 207
12.16 Fragment of the Input Sangam graph `G`. 209
12.17 The Context Dependency Tree Produced by the Basic Inlining Technique for the Input Sangam graph in Figure 12.16. . 209
12.18 The Resultant Fragment of the Output Sangam graph `G'` Produced by the Evaluation of the CAT in Figure 12.17 Defined on the Input Sangam graph in Figure 12.16. 209
12.19 Fragment of the Input Sangam graph `G`. 210
12.20 The Context Dependency Tree Produced by the Basic Inlining Technique for the Input Sangam graph in Figure 12.19. . 210
12.21 The Resultant Fragment of the the Output Sangam graph `G'` Produced by the Evaluation of the CAT in Figure 12.20. . . . 210
12.22 Fragment of the Input Sangam graph `G`. 212
12.23 The Cross Algebra Graph Produced by the Inlining the Path `p:e1.e2.e3` as shown in Figure 12.22. 212
12.24 The Resultant Fragment of the the Output Sangam graph `G'` Produced by the Evaluation of the CAT in Figure 12.23. . . . 212

12.25 A Fragment of the XMark Benchmark DTD as shown in Figure 10.2 Depicted as a Sangam Graph (Sangam graph). No order and quantifier annotations are shown for the back-pointer edges to distinguish them in the figure. These are defaulted to `1` and `[1:1] respectively`. 213
12.26 The Cross Algebra Graph that represents the Basic Inlining Technique applied to the Sangam graph in Figure 11.3. This Sangam graph is only for the node with root `item`. Cross Algebra Trees similar to the ones given in this figure will be produced for each root node. 214
12.27 Output Sangam graph Produced by the Evaluation of the Cross Algebra Graph in Figure 12.26 for Input Sangam graph in Figure 12.25. 215
12.28 The Relational Schema Produced by the Translation of the Sangam graph in Figure 12.27. 215

13.1 A Cross Algebra Framework for Relational and XML Models. 220
13.2 Loadin Time. 224
13.3 Load Time vs. XML Size. 225
13.4 Generation Time. 226
13.5 Load vs. Generation. 227
13.6 Cross: Evaluation Time vs. Input Size. 228
13.7 Connect: Evaluation Time vs. Input Size. 229
13.8 Smooth: Evaluation Time vs. Input Size. 230
13.9 Subdivide: Evaluation Time vs. Input Size. 231
13.10 Performance Comparison of the Four Algebra Operators. . . 232
13.11 Ident and Inline: Evaluation Time vs. Input Size. 233
13.12 Ident and Inline: Evaluation Time vs. Input Size. 234

14.1 An Example XQuery Statement to Insert Data Values into an XML Document. 253

15.1 First Pass of Cross Algebra Tree Target Maintenance Algorithm. 267
15.2 First Pass of Cross Algebra Tree Target Maintenance Algorithm - The UpdatePair Function. 268
15.3 A Cross Algebra Graph. 272
15.4 The Updated Cross Algebra Graph After Step 1. 274
15.5 Second Pass - Clean Up of Cross Algebra Tree Target Maintenance Algorithm. 275

15.6 An Example CAG. 277
15.7 Example: Modified Input Sangam graph G_u produced by Application of insertNode (D, τ, A, el, 1:1) on the input Sangam graph G in Figure 15.6. 278
15.8 Example: The New CAT CT3 Produced by insertNode and insertNodeAt Propagation Steps. 278
15.9 Example: Modified CAG CAG' after the Addition of the new CAT CT3. 279
15.10 Example of Input Sangam graph G. 280
15.11 Example of Modified Input Sangam graph G_u. 280
15.12 Example of The New CAT Produced by insertEdge and insertEdgeAt Propagation Steps. 280
15.13 Another Example of a Cross Algebra Graph. 286
15.14 A CAT Example that maps the input Sangam graph G to the output Sangam graph G'. 289

16.1 The personal.dtd, used as the input DTD. 310
16.2 Incremental Propagation vs Recomputation for Cross Operator. 313
16.3 Incremental Propagation vs Recomputation for Connect Operator. 314
16.4 Incremental Propagation vs Recomputation for Smooth Operator. 315
16.5 Incremental Propagation vs Recomputation for Ident Expression. 316
16.6 Incremental Propagation vs Recomputation for Inline Expression. 317
16.7 Incremental Propagation for Cross Operator. 318
16.8 Incremental Propagation for Connect Operator. 319
16.9 Incremental Propagation for Smooth Operator. 320
16.10 Incremental Propagation for Smooth Operator. 321
16.11 Execution Time Comparison of deleteObject Propagation. 322
16.12 Execution Time Comparison of insertEdgeObject Propagation. 322
16.13 Execution Time Comparison of deleteEdgeObject Propagation. 322
16.14 Incremental Propagation through an Ident Expression. . . . 324
16.15 Incremental Propagation through an Inline Expression. . . . 325
16.16 Time Comparison for insertObject Propagation. 325
16.17 Time Comparison for deleteObject Propagation. 326

16.18 Time Comparison for insertEdgeObject Propagation. . . 326
16.19 Time Comparison for deleteEdgeObject Propagation. . . 327
16.20 Propagation Time of insertObject Vs. Number of Algebra Operators. 328
16.21 Propagation Time of deleteObject Vs. Number of Algebra Operators. 329
16.22 Propagation Time of insertEdgeObject Vs. Number of Algebra Operators. 329
16.23 Propagation Time of deleteEdgeObject Vs. Number of Algebra Operators. 330

17.1 Schema in the XAlgebra Type System. 350
17.2 Data shown in the Type System of XAlgebra. 350
17.3 Projection in XAlgebra. 351
17.4 Projection in Relational Algebra. 351
17.5 Projection in Cross Algebra. 351
17.6 Iteration in XAlgebra. 353
17.7 Iteration in Cross Algebra. 353
17.8 Schema and Data for Reviews element in XAlgebra Type System. 354
17.9 Join in XAlgebra. 355
17.10 Join in Relational Algebra. 355
17.11 Join in Cross Algebra. 356

A.1 The auction.dtd of the Xmark Benchmark [SWK+01]. . . 374
A.2 The auction.dtd of the Xmark Benchmark [SWK+01]. . . 375
A.3 The auction.dtd of the Xmark Benchmark [SWK+01]. . . 376
A.4 The auction.dtd of the Xmark Benchmark [SWK+01]. . . 377
A.5 The personal.dtd used for Sangam testing. 378
A.6 The play.dtd available with JAXP1.1. 379

Part I

Information Integration and Change Management

Chapter 1

Introduction

1.1 Issues in Change Management

Today, databases are used as persistent stores for applications such as e-commerce, multi-media or large design applications. These applications are often extremely volatile in nature. This volatility is due in part to the change in user requirements, a fix to an erroneous condition, or a need to support new applications. All of these requirements can manifest themselves in the database as changes in the data, structure, constraints, permissions or rules. Change in the data values has been recognized as a routine problem and is handled in most commercial systems [KL95, Tec94, Tec92]. For schema (structural) changes alone, Sjoberg [Sjo93] has documented an increase of 139% in the number of relations and an increase of 274% in the number of attributes, and change in syntax of every relation in the schema at least once during the nineteen-month period of the study. This study was done in the development and initial phase of a health management

system at several hospitals. Other changes such as the changes in structure, rule, constraints and model are handled to varying degrees of sophistication and completeness (Figure 1.1) [RAJB00]. The workshop *Evolution and Data Management* workshop at *Conference of Conceptual Modeling (ER), 1999* has documented some of the research on the evolution in data, rules, constraints, models, and meta-models.

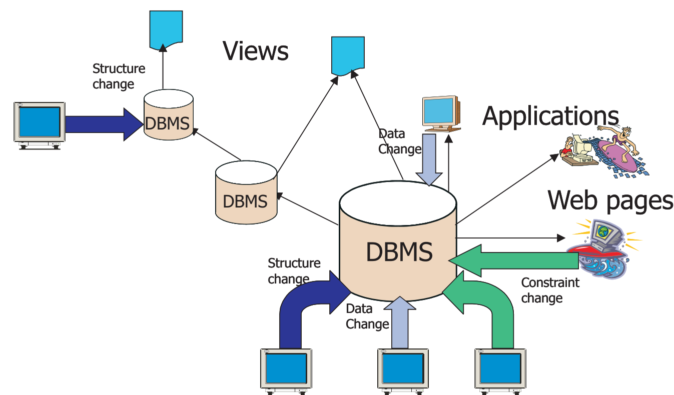


Figure 1.1: Changes in the Database Environment.

For databases to be effective persistent stores, they need to provide along with other functionality, comprehensive support for change management. Database management systems must thus address the following two key issues:

- **Change Specification**: First and foremost, a user must be able to specify and execute a change on a set of information (a database system). The database system must therefore adequately address questions such as: (1) How does a user effectively specify a change in the database? and (2) How is that change executed in the database system

while ensuring that all information will be correct after the execution of the change?

- **Managing Effects of Change on Other Parts of the System**: Information rarely exists in isolation. More often than not, applications are written that operate on the information and produce reports, web pages or XML documents, or subsets of large information sets are defined to make the data sets more manageable, or information is restructured and presented to the user in other formats, i.e., in other data models. Such subsets of information are said to be derived from the base information. A change specified by a user must therefore be executed not only on the local information, but also its affect on all information sets that derive from it must be managed. The database system must therefore adequately address questions such as: (1) What are the best techniques for propagating a local change to the derived information? and (2) Is the derived information still valid?

1.2 Change Management - State of Art

1.2.1 Change Specification and Execution

Restructuring or change specification support in database systems generally implies changes to the stored data, the structure, the methods that are defined in the type (the behavior), as well as the maintenance of constraints and rules[RAJB00]. While today most systems provide constraints, rules etc., change support exists only for data updates and structural evolution

such as the addition or deletion of a type [BKKK87, Tec94, Bré96, KC90, SZ86]. Limited support exists for the evolution of behavior. Research typically has not looked at issues such as the maintenance or evolution of constraints defined for the database in the event of a change, or in depth at the provision for more complex forms of schema changes such as the merging of two classes or splitting a class into two or more classes [Bré96, Ler96], beyond the simple changes applied to individual types such as the `addition`, `deletion` and `modification` of attributes.

Data changes are perhaps the most common type of change and have been comprehensively studied in literature. In fact, all commercial systems [KL95, Tec94, Tec92, Tr00, BMO+89, Obj93, BKKK87, Inc93] provide support to add, remove or modify information. In most cases this ability to manipulate the information is built into the query language that can also access the data. These are termed *data changes* and are a de-facto standard [ANS92].

Schema evolution or structural changes is another active area of research. In this dissertation we primarily focus on structural changes. We now present a more in-depth discussion on the state of art and present some active research challenges in this area.

For object-oriented databases, some research has been done on the maintenance of behavior, i.e., the methods that are defined for a class, in the event of structural change [MNJ94, OPS+95, BH93, LZLH94]. Behavioral evolution is an active area of research both in the area of databases as well as in software engineering. We do not address this issue but do give a brief synopsis of some active work in this area in Chapter 9.

Structural Changes

Change Specification. Most commercial database systems [KL95, Tec94, Tec92, Tr00, BMO$^+$89, Obj93, BKKK87, Inc93] also provide some support for enabling structural changes. This support is generally termed restructuring support or schema evolution. In most commercial systems schema evolution specification is supported by a *pre-defined* taxonomy of *simple fixed-semantic* operations. However, such simple changes, typically limited to individual types, are not sufficient for many advanced applications [Bré96]. More radical changes, such as combining two types or redefining the relationship between two types, are either very difficult or impossible to achieve with current commercial database technology [KGBW90, Tec94, BMO$^+$89, Inc93, Obj93]. In fact, most systems require the user to write *ad hoc* programs to accomplish such transformations. In the last few years, research has begun to look into the issue of complex changes [Bré96, Ler00]. Breche [Bré96] and Lerner [Ler00] both provide a *fixed* set of some selected, more complex operations.

The provision of any fixed set, simple or complex, is not satisfactory, as it would be difficult for any one user or system to pre-define all possible semantics and all possible transformations that could ever exist. It is impossible to predict all transformations that a user may desire. This problem exists for simple transformations but becomes even more pronounced for complex transformations. For example, consider the merging of two source classes into one single merge class. The structure of the new merge class can be defined in many different ways such as performing a union, inter-

section or a difference of the properties of the two input classes. Moreover, there can be many different semantics for populating this merge class, such as a value-based join on some pair of properties, a join based on a unique identifier, etc. Similarly, multiple choices exist for the fate of the source classes and the placement of the merged class in the class hierarchy. The more complex the transformation, the more difficult it becomes to predict all possible semantics to be desired in the future.

Clearly to handle the volatility in structural changes a fixed taxonomy of changes, simple or complex, is simply not adequate. Previous research has looked at extending the fixed taxonomy by providing some measure of user-flexibility. Shu *et al.* [SHL75] introduced a new data translation language, CONVERT, for translating between source items and target items. Davidson et al. [DK97] have defined a new language, WOL, for specifying the database transformations, while in O_2 [Tec94] and in work by Kim *et al.* [KGBW90] they have relied on C++ and C programming languages to provide this measure of flexibility. However, while these approaches offer extensibility, i.e., they allow users to modify and add new transformations, they provide only limited re-usability and portability as the transformations are specific and often cannot be shared across applications.

Challenges. An active challenge for change specification is to provide a principled approach that provides user flexibility and extensibility to handle the specification of structural changes. Ideally, such an approach would be independent of platform, database vendor and application domain. And while not an essential criterion, from a software perspective it would be ideal if such an approach could be easily integrated with existing database

systems.

Schema Evolution Correctness. A key criteria for schema evolution is the ability to guarantee that the database after the evolution is *consistent.* Banerjee *et al.*[BKKK87] state that a schema is consistent if it preserves all the invariants of the data model. Banerjee *et al.* [BKKK87] defined consistency and correctness of their schema evolution primitives in the context of the Orion system. Similar consistency and correctness of schema evolution primitives is provided by most commercial systems [KL95, Tec94, Tec92, Inc93].

Challenges. A key challenge is to address the issue of correctness with respect to complex operations. Like the primitives, the complex operations must also ensure that the database is not corrupted after their application. If users are allowed to specify their complex changes, a related challenge is to provide some notion of a user-level consistency, as is often provided in software systems. A user-level consistency would allow the writer of a complex operation to specify a set of schema-level and data-level constraints that must be satisfied after the complex operation execution.

Optimization. Schema evolution in general is an expensive process both in terms of system resource consumption as well as database unavailability [FMZ94b]. Researchers have approached improving system availability during schema evolution by proposing execution strategies such as deferred execution [Tec94, FMZ94b]. Kahler *et al.* [KR87] have looked at pre-execution optimization for reducing the number of update messages that

are sent to maintain replicated sites in the context of distributed databases. In their approach, the messages are simple data updates on tuples. They sort the number of messages by their tuple-identifier, and then condense (with merge or remove) the change history of the tuple into one update operation.

Challenges. With complex operations, it is not clear that these approaches or techniques would be directly applicable to optimize the execution of complex operations. A challenge in this area is to extend existing optimization techniques or propose alternative techniques to increase the database availability when complex operations are applied to it.

1.2.2 Managing the Effects of Change

Often due to the quantity of the information, data is partitioned, restructured and presented to the user in the form of views defined over one or many sources. Or conversely information from several systems is integrated into one database. Today, most database systems support *views* and *view schemas* defined over either one source or multiple sources [Obj93, Tec92, Tec94, Run92, BCGMG97, Day89, HD91] to support manageable, specialized sets of information for large enterprise systems. This information is often dispersed over multiple tiers in a combination of physical (source) and virtual (view) databases in an effort to service a large community of users [RS99a]. Design systems are an example of large-scale systems that have to service the needs of many users, often hundreds of users [PMD95]. In [PMD95], MacKellar and Peckham describe how a large-scale design is decomposed into a number of specialized tasks each requiring

its own representation of the design. Users often specialize in one aspect of the design and thus only deal with one representation of the design, also termed a perspective or a view. In such large often multi-tier systems, views (virtual databases) are built either directly upon the source database systems or upon another tier of derived information. Any change on any one source affects the many views that may be defined over it.

Research has approached this problem of managing *derived information* in the event of a source change from different angles. The simplest approach but a rather expensive approach is to recompute the views. Another approach is to incrementally *push* the change from the source to the view allowing the view access to the change in an immediate manner. This approach is often utilized for data updates. Several view maintenance algorithms [GB95, BCGMG97, SLT91, San95, KR98, AYBS97] have been proposed that handle the incremental propagation of data updates from the source to the view. Much of this work exists in the context of the relational [GB95, KR98] and the object data models [BCGMG97, SLT91]. There are some proposed variations to these basic strategies with respect to the timing (deferred, immediate, or at fixed intervals) of propagation and recomputation.

However, these approaches do not focus on structural changes. Effectively managing the effect of structural change has been an active area of research for some time. Some of the proposed techniques to handle these changes are versioning and view mechanisms that hide the change, or adapt the derived information. Versioning [Lau97b, Lau97a, KC88, MS93, SZ86] creates a new version of the entire database in some cases every time

a change occurs. Proposed view mechanisms [Kau98, RR97] utilize view technology to make the schema change transparent by re-writing the other dependent views. There has also been work to adapt affected views by using system-available redundant information [RLN97a, NLR98, RLN97b] as well as in making the view definition language itself resilient to the changes in the underlying source [Har94].

Challenges. Managing the effects of schema change is an active area of research with many un-addressed research issues. Here we list some of the open issues.

- Object-oriented Views - Limited support exists for managing the effect of structural changes in object-oriented views, especially object-generating views.
- XML Views - Much of the work that we list here and in the related work chapter (Chapter 9) has been presented for relational or object-oriented views. Limited work [TIHW01, NACP01, QCR00] exists for managing change, both data and structural, in XML views.
- Complex Changes - Current research focuses on providing support for managing simple, primitive changes [RLN97a, NLR98, RLN97b, GB95, BCGMG97, SLT91, San95, KR98, AYBS97]. However, systems such as O_2 exist that now support complex changes. Although O_2 does support views, they do not have any support for the propagation of complex changes to the views. For database systems to provide comprehensive support for managing change, they must provide support for handling complex as well as simple changes.

- Across Data Models - Almost all of the current work on managing the effects of change on views exists within one data model, i.e., the source and the view are in the same data model. Limited work exists [ZLMR01, TIHW01] attempts to cross the data model boundary. With the ever expanding need to store XML data in relational, extended relational or object databases, this maintenance that crosses data model boundaries has become a critical issue and one of great significance. This is a key focus point of this dissertation and thus we now expand on this issue and present related sub-issues below.

Cross Data Models and Integration

Data management over the years has matured from hard-to-maintain specialized file management systems to a generic simple model of data in commercial relational databases to a more complex object model in object databases (ODB) to yet again a more flexible semi-structured XML data model. With this evolution of data models has come the need to integrate information from a heterogeneous set of data sources, as well as to translate information in one data model to information in another data model. For example, with the XML model, a bulk of the research effort has focused on translating XML to existing, established relational [ZLMR01, FK99, SHT+99], object relational [SYU99] or object-oriented database systems [CFLM00] to store and manage XML. This effort has led to many special-purpose transformations that have been proposed in the last few years to handle the mapping of XML into relational systems, object-relational and object systems [ZLMR01, FK99, SHT+99, CFLM00, SYU99].

While much effort has concentrated on data integration, managing the effect of source change when the target or the view is in a different data model is a problem that is still in its infancy. A solution beyond recomputation that has been looked at is the incremental propagation of the change from the source to the target. Tatarinov *et al.* [TIHW01] have presented data update primitives and an algorithm that propagates data changes from XML to the relational model. Similar work has also been done by Zhang *et al.* [ZLMR01] which also handle the propagation of schema changes from XML to the relational model. The propagation of the change in both of these approaches is tightly coupled to the actual algorithmic mapping of the source information to the target. A change in this mapping algorithm would necessitate a modification in the re-translation of all data and schema changes. For example, many algorithms such as basic inlining [STZ+99] or shared inlining [STZ+99] have been proposed to transform the XML model to the relational model. For each algorithm we would need to provide a set of translations that map an XML change to an equivalent change on the relational model. Any change in the algorithm would require a re-translation of the set of changes.

Challenges. We must address several issues to provide the same degree of capabilities that exist for managing the effect of change in views within the same data model as the source to now manage the effect of a change on the view in a different data model. The primary issue is the necessity to have a principled solution to handle the translation of one data model to another. With such a principled solution in place, generic propagation algorithms must be developed to enable the propagation of change across

data model boundaries. More sophisticated techniques such as versioning [Lau97b, Lau97a, KC88, MS93, SZ86] or view mechanisms to *hide* the change [Kau98, RR97] must also be re-visited in this new context. These remain open challenges in this area.

1.3 Our Work

This dissertation addresses these two key research issues in change management - change specification (Issue 1.2.1) and managing the effect of change (Issue 1.2.2). More specifically we address the following issues.

- **Change Specification:**

  1. Providing a principled approach to enable user flexibility and extensibility when specifying structural changes.
  2. Providing consistency and correctness checks for execution of complex changes.
  3. Providing optimization of complex changes.

- **Managing the Effects of Change:**

  1. Providing a principled approach for specifying the mapping of information in one data model to another.
  2. Providing a generic algorithm to propagate a source change to the derived information, independent of the data models (source and target) and the actual translation of information from the source to the target.

3. Providing at a specific level the propagation of a change from a relational source to an XML target, as well as the propagation of a change from an XML source to a relational target.

1.3.1 Change Specification - The SERF Framework

The Basic SERF Framework

To address the limitation of *current* schema evolution technology, we propose in this dissertation the SERF framework [CJR98c, CJR98a] that allows users to perform a wide range of complex user-defined schema transformations *flexibly*, *easily* and *correctly*. Our approach is based on the hypothesis that complex schema evolution transformations can be broken down into a sequence of basic evolution primitives, where each basic primitive is an invariant-preserving atomic operation with fixed semantics provided by the underlying system. To effectively combine these primitives and perform arbitrary transformations on objects within a complex transformation, we propose to use a query language. Previous research has resorted to using a programming language to achieve *ad hoc* user transformations [KGBW90] or defining a new language [DK97] for specifying the database transformations. In our work, we propose the use of the standard query language for object database systems, OQL [Cea97], and demonstrate it to be sufficient within our framework if combined with meta-data access. One drawback of this approach is the coupling of SERF with the ODMG Object Model [Cea97]. While in principle this approach could be translated to other systems, the underlying assumptions as given in Chapter 4 must

be re-investigated.

SERF transformations, *ad hoc* programs, and the use of new languages all suffer from the fact that they specify the transformation for a particular schema. In SERF, we go one step further by introducing the concept of a SERF *template*. A template extends the notion of a SERF transformation to be a named transformation that can include variables and input and output parameters. The SERF transformation code itself is written to be generic, that is not bound to particular schema elements, and can be applied based on the provided input parameters. A template can thus be applied to different schemas. Furthermore it can also be re-used for building more complex transformations. These generic templates can also be applied to different systems, i.e., for different object databases and different object models, thus making them a valuable community wide resource. Thus, one of the goals of this work is to provide a library of these restructuring templates for different domains as a resource for restructuring and transforming of data.

Soundness and Consistency

Guaranteeing correct semantics of all schema evolution operations, i.e, ensuring that they produce as output a schema and data that both conform to the invariants of the underlying system, is key. In our work we provide two levels of consistency. The first is termed *invariant preserving* consistency. The invariant preserving consistency ensures that after the execution of a SERF template, the resultant OODB database conforms to the invariants of the data model. We show that that if each schema evolution operation is an invariant preserving operation, then a SERF template is also invariant

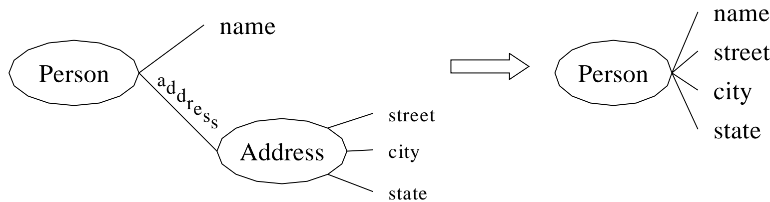


Figure 1.2: The Inline Operation.

preserving [CRed].

While a SERF template may be indeed invariant-preserving, its inherent complexity may lead to changes in the schema and data which, while invariant preserving, may not be desirable. For example, consider the example of *inlining* shown in Figure 1.2. Here, all the attributes of the class `Address` are inlined into the class `Person`, and subsequently the class `Address` is deleted. Now assume that in the class `Person` there exists a self-referential relationship `spouse` that refers back to the `Person` class. The inlining of the relationship `spouse` would result in the deletion of the class `Person`, a clearly undesirable consequence of the operation. To allow the identification of such semantics when applying a SERF template, we define ***template semantic*** consistency. To now enable users to specify template-semantic constraints, we introduce the notion of ***Template Wrappers*** [CRH00b]. These Template Wrappers, based on Software Contracts [Mey92], allow a user to specify semantic constraints on a template which can then be checked at runtime or prior to execution [CRH01].

Optimization

A third important aspect of schema evolution is its execution time. Schema evolution in general is an expensive process both in terms of system resource consumption as well as database unavailability [FMZ94b]. Even a single simple schema evolution primitive (such as add-attribute to a class) applied to a small database of 20,000 objects (approx. 4MB of data) has been reported to take about 7.4 minutes [FSS+97]. An *inline* operation on a database of 20,000 objects applied on the example given in Figure 1.2 will take about 25 minutes. We thus look at reducing the execution time of verified templates. For this we present an approach that looks at combining operations, or eliminating operations to reduce the number of operations that are to be executed [CNR00]. In order to validate our approach, we have conducted a set of experiments that confirm that our optimization heuristics, when applied to a sequence of operations, greatly reduces the total evaluation time.

Validation of SERF Concepts

Case Study. As a step towards validating the SERF framework, we present a case study of the complex schema evolution operations we have found in the literature. We have also applied the SERF framework as a tool for website restructuring. We highlight some of the templates used for this. We have also explored the utilization of the SERF framework to actually aid in the software evolution of a schema evolution facility.

Implementation. To further validate our proposed concept of SERF transformations, we have developed a working system, called OQL-SERF. OQL-SERF [$RCL^{+}99$] serves as a proof of concept and helps explore the suitability of the ODMG standard as the foundation for a template-based schema evolution framework. The OQL-SERF development is based on the ODMG standard, which is a reliable basis on which to develop open OODB applications. The ODMG standard defines an Object Model, a Schema Repository, an Object Query Language (OQL) as well as a transaction model for OODBs (see Section 3). OQL-SERF uses a subset of the ODMG 2.0 standard. It uses an extension of Java's binding of the ODMG model as its object model, our binding of the Schema Repository for its Meta-data Dictionary and OQL as its database transformation language. However, the ODMG standard does not define any evolution support for its object model. Thus, as part of our effort we have defined the invariants for preserving the ODMG Object Model and also a set of schema evolution primitives that preserve these invariants.

OQL-SERF is built as a thin evolution layer on top of Objectstore's persistent storage engine PSE [O'B97]. As part of our implementation we have developed a schema evolution facility for PSE.

1.3.2 Managing the Effects of Change - Sangam

The Cross Algebra

One significant contribution of the database community has been the development of query languages and query algebra which today are considered

a de-facto standard. These query languages and algebras define subsets of information, or restructure information before presenting them to the user. In other words they are used to translate one set of information to another set within the same data model. Achieving the same translation across data model boundaries however has led to the propositions of several algorithms that can translate information in one data model to information in another data model. Translations of this form have several disadvantages. First, the standard approaches to optimization, for example query optimization, are not generally feasible. Second, maintenance, i.e., translation and propagation of a change in one data model to a change in the other data model is not achievable in a generic manner. Finally, notification services of any kind are not possible without extra binding information between the source and the target. In order to accomplish these *tools* in a generic manner, we must therefore first tackle the problem of making generic the mapping between the data models. That is, we must first define a generic mapping language to describe the mapping between two schemas that potentially belong to two different data models.

However, as has been noted by researchers [RR94] developing a mapping language a la query algebra to go across data models is hard, if not an impossible problem. A more traditional approach is the middle-layer approach, where in information from the local sources is translated into a common data model. Translations from one data model to another can then be accomplished by performing translations or mappings in the middle layer.

In this dissertation, we follow this middle-layer approach to address

the problem of a generic mapping language between data models. To accomplish this we make two contributions: (1) the description of a common data model - the *Sangam graph model*; and (2) the description of a generic mapping language - the *cross algebra*.

In choosing a common data model for our work, we wanted the model to (1) be expressive enough to structurally represent schemas from a variety of different data models such as the relational, XML or object models; and (2) be able to express a common subset of constraints, such as the order constraints in XML, participation constraints (relational and XML), and other referential constraints such as key and foreign key constraints. Existing, off-the-shelf data models were the most attractive choice for a common data model, as they provide considerable advantages in terms of existing tool-sets and a user-base. However, we found that the existing data models did not satisfy the requirements of expressiveness and common subset of constraints that we had laid out. For example, while the XML model satisfies the expressiveness property, it does not provide adequate support for key and foreign key constraints [1]. Similarly, the relational and the object model do not support order constraints.

In our work, we have thus chosen to define a new common data model - the *Sangam*[2] *graph model*. The Sangam graph model is based on the common denominator of the existing data models - a graph, and can represent a subset of the common constraints present in existing data models. The

[1]The XML Schema specification given in May 2001 does provide support for keys and keyrefs. However, the work in this dissertation was already under-way and towards completion at that point.

[2]*Sangam* is a Hindi word meaning “the union”.

Sangam graph model thus is a simple graph based model that can express schemas from different data models, including the XML, relational and object models. It can also capture order, participation, key and foreign key constraints.

To accomplish transformations in the middle-layer, we have defined a new *transformation language*, the *cross algebra*, that operates on Sangam graphs. The cross algebra covers the class of linear transformations [GY98] applied to a graph that represents schemas from different data models. It is composed of the primary graph operations such as the adding of a node or an edge (`cross` and `connect` nodes), combining two edges (`smooth`) and splitting an edge (`subdivide`). Each cross algebra operator can translate only an individual schema entity.

To enable the translation of an entire schema in one data model to a schema in another data model, we also allow the composition of these algebra operators. We provide the traditional composition by derivation, i.e., a derivation tree composed of cross algebra operators. However, unlike other algebras, relational, XML, or object, that can define a fixed granularity of a modeling unit (a relation is a modeling unit for the relational algebra), an algebra that crosses different data models must be flexible in order to accommodate the variance in the sizes of *modeling units* of each data model. As the modeling granularity is distinct in different data models, we use the smallest granularity as a modeling unit. This has the advantage of allowing the mapping of constructs and relationships in one data model to constructs and relationships in another data model. However, given this low-level granularity we need additional mechanisms to enable us to ex-

press the mapping of complex modeling constructs such as a relation or a complex nested XML element from the input data model to the output data model. To enable this, we introduce a new type of composition, *context dependency*, of cross algebra operators that allows several algebra operators to collaborate and jointly operate on disparate sets of modeling constructs and together produce one connected complex output construct. Information in one data model can be translated to another data model via cross algebra expressions composed of cross algebra operators connected by derivation or context dependency. In this dissertation, we also present the evaluation algorithm for executing a cross algebra expression. We show that the evaluation algorithm (1) terminates, and (2) it produces a valid output.

The cross algebra operators can currently express the majority of the translation algorithms found in literature [ZLMR01, FK99, SHT+99, CFLM00]. Moreover, the cross algebra operators are independent of the source and target data models. To validate our proposed ideas we have implemented a prototype system and conducted several experiments that (1) validate in practice that we are indeed able to express a variety of translation algorithms; and (2) give a measure of the performance of the prototype system.

Propagation of Change

With the mapping of application schemas described by the cross algebra graphs, we can now focus on our key goal, i.e., to propagate a local change on the source schema to the target schema. To achieve this desired propagation of update from the source to the target, in this dissertation we present two incremental propagation algorithms, *Gen_Propagation* and *In-*

sert_Propagation that can handle a set of common schema evolution and data update operations. These propagation algorithms are (1) independent of the source and target data model; and (2) are loosely-coupled to the translation between source and the target data models. The data model independence allows us to apply these algorithms for managing the maintenance of targets irrespective of whether the source and the target is in XML, relational or object model. The loose-coupling to the translation between the data models allows us to still utilize the mapping during the propagation step. However, because of our strategy the actual translation of a change from the source to the target is not affected as a result of a change in the mapping.

A key criteria for incremental update propagation is to ensure that the output produced by the application of the update sequence (produced during update propagation) is the same (identical) to the output that would be produced if the modified cross algebra were to be re-evaluated completely. We show formally that our propagation algorithms *Gen_Propagation* and *Insert_Propagation* can achieve this *equivalence*.

The goal of incremental propagation is to propagate a change from the source to the target in an efficient manner, i.e., it should provide a better response time than the complete re-evaluation of the modified cross algebra graph. We experimentally show that the incremental propagation provides a better response time than recomputation.

Validation

Maintenance of Relational Data and XML Documents. An objective of this research was to enable the maintenance of the XML views defined over relational data, as well as the maintenance of relational views over XML data, irrespective of the translation utilized. Clearly, one option was, and still is, to produce hard-coded algorithms where each algorithm represents a combination of one update operation such as the deletion of an element in a DTD [SKC+01] and one mapping technique such as the basic inlining technique [STZ+99]. An alternative was to use the cross algebra, as outlined above, to represent the mapping of the relational source to the XML view or vice versa and to then apply the *Gen_Propagation* and *Insert_Propagation* propagation algorithms to achieve the maintenance of the views in case of a source change. We believe that the second alternative offers many advantages as outlined above, and hence we use cross algebra to represent the mapping of XML to relational. The application of this approach also serves as a primary validation of our cross algebra and propagation algorithms.

In this dissertation we show how cross algebra graphs can be defined between the XML source and the relational target. In particular we show how the basic and shared inlining [STZ+99] techniques can be represented as cross algebra graphs. For both of these cross algebra graphs, we show how a change (DTD change or an XML document change) can be translated and propagated through the cross algebra graphs using the *Gen_Propagation* and *Insert_Propagation* algorithms. The update sequence produced by this

incremental propagation can then be applied to the output to achieve the source-equivalent change.

Implementation. To further validate both our cross algebra and the propagation algorithms, we have developed a working system, called 𝔖𝔞𝔫𝔤𝔞𝔪 [CRZ+01]. 𝔖𝔞𝔫𝔤𝔞𝔪 serves (1) as a proof of concept for the general framework of cross algebra that we have proposed here; (2) as a testbed for experimentally testing the power of the cross algebra graphs in terms of its modeling capability with respect to XML and relational models, and the different mapping techniques in literature [STZ+99]; (3) as a testbed for experimentally validating the ability to propagate a change from XML to relational and vice versa with correct results; and (4) as a testbed for experimentally proving that incremental propagation is faster when comparing user response time than complete re-evaluation for both basic and shared inlining cross algebra graphs between XML source and relational targets.

The complete implementation for the 𝔖𝔞𝔫𝔤𝔞𝔪 system has been done in Java JDK 1.4 and different third-party Java packages [Wut01, Sys01, Jav96].

1.4 Organization of this Dissertation

This dissertation is organized into four parts. Part I includes this introduction that reviews the general issues in restructuring and transformation of information.

Part II describes the SERF framework. In Chapter 3 we present the necessary background in Chapter 3. Chapter 4 gives the details of the basic

SERF framework. We discuss the soundness and consistency of templates in Chapter 5. In Chapter 5.3 we show how SERF templates with contracts can be utilized to verify a SERF template prior to execution. Chapter 6.1 discusses the optimization strategies that we have developed to reduce the time of execution of a SERF template. Chapter 7 discusses the working implementation of the basic SERF system. Chapter 8 presents a case study of the complex transformations found in literature as well as the applicability of SERF to Web-site restructuring. Chapter 9 gives an overview of the related work in this area.

Part III focuses on the algebra for transforming information across data model boundaries and their subsequent maintenance. Chapter 11 describes the graph model (Sangam graph) that we use as our basic data model. Chapter 12 describes the cross algebra and the different techniques of composing them into cross algebra graphs. Chapter 13 describes the working implementation of the Sangam system and presents the experimental validation of the same. In Chapter 14 we present the set of change operations that can be applied on a Sangam graph, and show how local changes in the relational or XML model can be mapped into these. Chapter 15 describes our incremental update propagation algorithm that can propagate these graph changes through the cross algebra graphs. Chapter 16 presents our experimental results and Chapter 17 gives an overview of the related work.

Finally, Part IV concludes this dissertation. Chapter 18 summarizes the results of this dissertation and presents some possible future extensions of this work. Appendix A lists the set of DTDs used in the experiments.

Part II

SERF - An Extensible Transformation Framework

Chapter 2

Overview

Support for complex changes in databases exists in the form of a set of pre-defined change operations that can be invoked with different parameters [Bré96, Ler00]. This work [Bré96, Ler00] defines a set of high-level primitives such as *merge*, *split* and *inline* for object-oriented databases, in particular for O_2. However, it is difficult to *a-priori* define (1) all possible complex operations; and (2) all the possible semantics for the set of complex operations. For example, a *merge* of two classes can be accomplished by combining the attributes and the extents of the two classes, or by forming a new class that contains only the common set of attributes for the two classes. For any change beyond the pre-defined set, a user must therefore write programs to manipulate the structure of the database as desired. Such an approach is error-prone, provides no guarantees for consistency of the database, does not lend itself to any kind of verification or optimization, and is not portable from one database to another.

In this part of the dissertation, we identify the fundamental components

of any complex change - the change expressed in terms of a set of primitive changes and the corresponding, potentially complex, data changes. Based on this hypothesis, we have developed SERF [CJR98c], an extensible and re-usable framework for schema evolution. In this part, we now present the details of this framework and other work that we have done in this general area (as outlined in Chapter 1).

Roadmap. In Chapter 3 we present the necessary background in Chapter 3. Chapter 4 gives the details of the basic SERF framework. We discuss the soundness and consistency of templates in Chapter 5. In Chapter 5.3 we show how SERF templates with contracts can be utilized to verify a SERF template prior to execution. Chapter 6.1 discusses the optimization strategies that we have developed to reduce the time of execution of a SERF template. Chapter 7 discusses the working implementation of the basic SERF system. Chapter 8 presents a case study of the complex transformations found in literature as well as the applicability of SERF to Web-site re-structuring. Chapter 9 gives an overview of the related work in this area.

Chapter 3

The ODMG Model and Schema Evolution

The SERF framework is based on the ODMG 2.0 standard [Cea97]. Here we give a brief description of the ODMG object model constructs that are pertinent to this work, and present our proposed primitive schema evolution support for the ODMG model.

3.1 ODMG Standard: The Object Model

The ODMG Object Model is based on the OMG Object Model for object request brokers, object databases and object programming languages [Cea97, Clu98]. For the purpose of SERF we limit our description of the ODMG Object Model to Java's binding of the object model. The most important impact this will have on our work is the restriction to single inheritance between types, which we will now assume for the remainder of this disser-

tation. Nonetheless, the Java binding of the ODMG is a powerful model.

Types and Objects. One of the basic modeling primitives of an ODMG-compliant database are *objects.* Each object has a unique object identifier which persists through the lifetime of the object and serves as a reference for other objects. All objects in the database are categorized by their *types* $\mathcal{T}$, i.e., a *type* $\texttt{t} \in \mathcal{T}$ defines the structure of an object and each object is an instance of some type in the database. The object cannot change its type in its lifetime. A *type* can define multiple properties (attributes), denoted by *N(t)* . Although ODMG defines a `property` as `attributes` or `relationships`, here we consider a `property` to be only an `attribute` as the Java binding of ODMG does not support relationships as yet.

While ODMG distinguishes between *types* and *classes*, for our work here we ignore the difference between them. The terms type and class are hence used interchangeably.

Inheritance. Although ODMG defines multiple inheritance, Java's binding of the ODMG Model supports only single inheritance. A subtype t_1 therefore inherits the range of states and behavior from its super-type *t.* Moreover, an object can be considered as an instance of its type as well as its super-type. The *native* properties *N(t)* refer to the properties of `type` *t* that are defined locally in type *t*. *Inherited* properties *H(t)* of a `type` *t* refer to the union of all the properties defined by all the (ancestral) super-types of `type` *t*[1]. We use the term $P_e(t)$ to refer to all sub-types of the type *t*.

[1]There can be only one direct super-type.

Extent of Types. Although Java's binding of the ODMG model does not as yet support the notion of extents, we have found it to be a necessary extension to the binding. We denote the extent of a type as *E(t)*. This is the set of all direct instances of type *t*. We use the notation $E_e(t)$ to refer to the instances of the type *t* and the instances of all its subtypes.

Relationships The ODMG object model supports the notion of a reference attribute which defines a one-way association between two classes as well as the a bi-directional association wherein if class `A` refers to class `B` then class `B` must refer to class `A`. The user can define the cardinality of these references as one-to-one, one-to-many or many-to-many. To capture this notion of association, we use the *referential relationship* ($\longrightarrow$) that specifies when one type refers to another type; and a *bi-directional relationship* ($\longleftrightarrow$) that specifies a referential relationship and its inverse.

ODMG Schema A schema *S* for our purpose here is composed of a set of type definitions $\mathcal{T}$ and a type lattice (tree) `T` such that $P_e(t)$ for all $t \in \mathcal{T}$ are included in `T`.

Table 3.1 summarizes the notation we use for describing the object model.

3.1.1 ODMG Schema Repository

A *Schema Repository* as defined by ODMG [Cea97] captures the database schema and its constraints as objects. The schema repository is used by the OODB at initialization time to define the structure of the database and at run-time to guide its access to the database. The schema repository is

Term	Description
S	The schema
$\mathcal{T}$, $\mathcal{C}$	All the types in the schema S
t	Elements of $\mathcal{T}$
$P(t)$	The immediate super-type of type t
$P_e(t)$	The union of direct and indirect super-types of type t
$C(t)$	The immediate subtypes of type t
$C_e(t)$	The union of direct and indirect subtypes of type t
$N(t)$	The native properties of type t
$H(t)$	The inherited properties of type t
$H_e(t)$	The union of of direct and in-direct inherited properties of type t
$E(t)$	The direct extent of type t
$E_e(t)$	The union of direct and indirect extent of type t

Table 3.1: Notation for Expressing the Object Model.

also accessible to tools and applications and hence SERF using the same operations that apply to user-defined types, like OQL.

The Schema Repository contains ***meta-objects*** interconnected by relationships that define the schema graph. A database schema together with the types and the properties of these types all exist in the schema repository as meta-objects. For example, a class `Person` and an attribute `name` are both meta-objects. Most meta-objects have a defining scope which gives the naming scope for the meta-objects in the repository. For example, the defining scope for `Person` is its defining schema and the defining scope for `name` is `Person`. In addition to this, the schema repository contains the relationships between the meta-objects that define the schema graph.

A class `Person` related to the class `Address` and the inheritance between two classes are examples of such relationships. These relationships help guarantee the referential integrity of the meta-object graph. Table 3.2 defines accessor functions used in the later sections.

Function	Return Type	Return Description
`t.localAttrlist`	Set	Local properties of type `t`
`t.scope`	Schema	Scope of type `t`
`t.metaClassName`	String	Name of type `t`
`p.attrName`	String	Name of property `p`
`p.attrType`	String	Domain type of property `p`
`p.scope`	Class	Scope of property `p`

Table 3.2: Commonly used Accessor Functions in the SERF Templates.

3.1.2 ODMG's Object Query Language - OQL

As part of its standard, ODMG has defined an object query language OQL that supports the ODMG data model. OQL is similar in format and features to SQL 92 but has extensions for some object-oriented notions such as complex objects, object identity, path expressions, polymorphism, operation invocation and late binding. In this section we describe a subset of the language that is used for the examples in the paper. For a complete description of OQL the reader is referred to [Cea97].

Selection. As a stand-alone query language, OQL supports the querying over *any* kind of object (i.e., individual object instances, collections and even the meta-data in the schema repository) starting from their names which act as entry points to the database. OQL supports querying with

and without object identifiers. For example, if the schema defines the types `Person` and `Employee` with extents `Persons` and `Employees` then it is possible to query `Persons` as follows:

```
select distinct x.age
from Persons x
where x.name = "Pat"
```

This selects the set of ages of all persons named Pat, returning a literal of type set<integer>.

```
select x
from Persons x
where x.name = "Pat"
```

This selects all persons with the name Pat, returning a literal of type set<Person> where each Person object in the resultant set has the same object identifier as that in the database.

Creation. OQL supports the creation of objects both with and without identity. For example, `Person(name: "Pat", birthdate: "3/28/95", salary:10000)` creates an instance of the type `Person` using the `Person` type constructor. This constructs a new `Person` object with a new object identifier. Similarly, `struct (name: "Pat" , birthdate: "3/28/95" , salary: 10000 )` yields a structure with three fields but no object identity.

Path Expressions. The ODMG object model as mentioned in Section 3.1 supports the naming of objects and also the reachability of other objects through this named object (i.e., persistence by reachability). From OQL, one therefore needs a way to *navigate* from a named object and reach the desired data. For example, the query `p.spouse.address.city.name` starts from a `Person`, gets her spouse (a `Person` again) goes inside the complex attribute of type `Address` to get the `City` object whose `name` is then accessed.

Method Invocation. OQL can call a method with or without parameters anywhere the result type of the method matches the expected type in the query. For example,

```
select p.oldest-child.address
from Persons p
where p.lives-in(``Paris'')
```

In this statement we retrieve the `address` of the `oldest-child` of all `Persons` who live in `Paris`. Here `oldest-child` is a method that takes no parameters and returns an object of type `Person`. The method `lives-in` is applied to the `Person` object and takes one parameter of the type `String` and returns `true` if the specific person lives in the target city.

3.2 Evolving the ODMG Object Model

Some support for schema evolution is provided by most OODBs [BKKK87, Tec94, BMO+89, Inc93, Obj93]. However, the ODMG 2.0 standard does not

yet address the issue of schema evolution. In this section we therefore describe a taxonomy of primitives to provide support for dynamic schema evolution of the ODMG object model based on a representative set found in other OODBs [BKKK87, Tec94, BMO+89, Inc93, Obj93]. We first present the invariants for preserving the ODMG object model and then the schema evolution primitives that preserve these invariants and hence the object model. The primitives presented here are *minimal* in that they cannot be decomposed into any other evolution primitives and *essential* in that they are all required for the evolution of the given object model. They can however be composed together with other evolution primitives to form more complex transformations, as we show in later sections.

3.2.1 Invariants for the ODMG Object Model

A schema update can cause structural inconsistencies. An important property imposed on schema operations is thus that their application always results in a *consistent* new schema [BKKK87]. The structural consistency of a schema is defined by a set of *schema invariants* of the given object model [Bré96]. In this section, we present the invariants for the ODMG object model adapted from the axiomatic model proposed by Peters and Ozsu [PO95].

Table 3.3 shows the notation we use for describing the axiomatic model. The *in-paths* and the *out-paths* are a set of pairs of `<type, name>`, i.e., a pair <c_1, r_1> where c_1 is the name of the class referring to *t* (or the class being referred to by *t*) and r_1 is the name of the reference attribute. The *in-degree* and the *out-degree* of a type is given by the count of all the types

Term	Description
types($\mathcal{C}$)	All the types in the system
S	The schema
$\mathcal{T}$	All the types in the schema S
$\mathcal{C}$	All the types in the schema S
t	Elements of $\mathcal{T}$
P(t), super(t)	The immediate super-type of type t
$P_e(t)$,*super* (t)*	The union of direct and indirect super-types of type t
C(t), sub(t)	The immediate subtypes of type t
$C_e(t)$, *sub* (t)*	The union of direct and indirect subtypes of type t
N(t)	The native properties of type t
H(t)	The inherited properties of type t
$H_e(t)$	The union of of direct and in-direct inherited properties of type t
E(t)	The direct extent of type t
$E_e(t)$	The union of direct and indirect extent of type t
in-paths(t)	The set of all paths referring to type t
in-degree(t)	The count of all paths referring to type t
out-paths(t)	The set of all paths going out of type t
H-out-paths(t)	The set of all inherited out-paths of type t
out-degree(t)	The count of all paths going out of type t
self-degree(t)	The count of all self paths of type t
H-out-degree (t)	The count of all *H-out-paths(t)* that are not self-referential
T-IN(t)	The total in-degree: *in-degree(t)* + *self-degree(t)*
T-OUT(t)	The total out-degree: *out-degree(t)* + *self-degree(t)* + *H-out-degree(t)*
$\mathcal{R}$	The set of all relations in the system

Table 3.3: Notation for Axiomatization of Schema Changes

other than itself referring to the type and vice versa. The *self-degree* is the count of self-references for a type. Thus the total in-degree, *T-IN*, of a type is given by the sum of the in-degree and the self-degree. Similarly, the total out-degree, *T-OUT*, is the sum of the out-degree, the self-degree and the

inherited out-degree.

The following invariants hold:

1. **Rootedness.** There is a single type $t \in \mathcal{T}$ that is the super-type of all types in $\mathcal{T}$. The type *t* is called the *root* of the type lattice ($\mathcal{T}$).

2. **Closure.** Every type in $\mathcal{T}$, excluding *root*, has a super-type in $\mathcal{T}$, giving closure to $\mathcal{T}$.

3. **Pointedness.** There are one or more types $\perp \in \mathcal{T}$ such that $\perp$ has no subtypes in $\mathcal{T}$. $\perp$ is termed a *leaf* of the type lattice.

4. **Singularity.** Every type $t \in \mathcal{T}$, excluding the *root*, has exactly one direct super-type $\texttt{t}_s$ in $\mathcal{T}$, i.e., $\mid P(t) \mid = 1$.

5. **Distinction.** Every type $t \in \mathcal{T}$ has a distinct name, i.e., $\forall$ $\texttt{t}_1$, $\texttt{t}_2$, if $\texttt{t}_1\texttt{.name} = \texttt{t}_2\texttt{.name}$, then $\texttt{t}_1 = \texttt{t}_2$. Every property $\texttt{p}$ for a type $\texttt{t}$ has a distinct name; the scope of name distinction for a property is the union of the inherited and native properties for type *t*.

6. **Degree.** The ratio of total in-degree, *T-IN* of the schema , to the total out-degree, *T-OUT* of the schema is an invariant.

3.2.2 Taxonomy of Schema Evolution Primitives

In this section we present the taxonomy of schema evolution primitives that we have designed for the ODMG object model such that they preserve the invariants introduced in Section 3.2.1 (Table 3.4). Our goal is to achieve a set of schema evolution primitives that is:

- Complete, i.e., our primitive set subsumes every possible type of structural schema change within the ODMG model.
- Minimal, i.e., none of the primitives can be achieved by a combination of two or more primitives.
- Consistent, i.e., each primitive is guaranteed to generate a valid schema as output when applied to a valid input schema. If the input schema is invalid, the operation is ignored and the schema is unchanged.

Completeness of Evolution Primitives. The taxonomy that we present here captures all changes needed to manipulate the ODMG type lattice as given in Section 3.1. We now outline a proof that shows that this set of changes indeed subsumes every possible type of schema change (completeness criteria). The proof we sketch has its basis on the completeness proof given by Banerjee et al. for the evolution taxonomy of Orion [BKKK87].

In order to show completeness of these operations we prove that every legal type lattice as defined by the invariants is achievable by combining the operations given in Table 3.4.

Lemma 1 *For any given type lattice* `T`*, there is a finite sequence of* `op2`*,* `op7` *and* `op9` *that can reduce the type lattice* `T` *to a type lattice* `T'` *consisting of only one single type, a root type.*

Proof: Assume that the type lattice has no relationships. It is apparrent if we repeatedly apply the operation `op2` which removes a leaf type `t`, we

Num	Evolution Primitive	Description	Error Condition
op1	*add-class*(c, $\mathcal{C}$)	Add new class c to $\mathcal{C}$ in the schema S	Class c already exists in S
op2	*delete-class*(c)	Delete class c from $\mathcal{C}$ in the schema S	c has sub-classes
op3	*add-ISA-edge*(c_x, c_y)	Add an inheritance edge from class c_x to c_y	there exists an edge from some class $\mathtt{c}_z$ to $\mathtt{c}_y$
op4	*add-attribute*(c_x, a_x, t, d)	Add attribute a_x of type t and default value d to class c_x and to all its sub-classes	$\mathtt{a}_x$ already exists in $\mathtt{c}_x$
op5	*delete-attribute*(c_x, a_x)	Delete the attribute a_x from the class c_x and removes it from all its subclasses	$\mathtt{a}_x$ does not exist in class $\mathtt{c}_x$
op6	*add-reference-attribute*(c_x, r_x, c_y, d)	Add unary relationship from class c_x to class c_y named r_x with default value d	$\mathtt{r}_x$ already exists in c_x
op7	*delete-reference-attribute*(c_x, r_x)	Delete unary relationship in class c_x named r_x	$\mathtt{r}_x$ does not exist in $\mathtt{c}_x$
op8	*form-relationship*(c_x, r_x, c_y, r_y)	Promote the specified two unary relationships to a binary relationship	$\mathtt{r}_x$ and $\mathtt{r}_y$ do not exist in classes $\mathtt{c}_x$ and $\mathtt{c}_y$ respectively
op9	*drop-relationship*(c_x, r_x, c_y, r_y)	Demote the specified binary relationship to two unary relationships	$\mathtt{r}_x$ and $\mathtt{r}_y$ are not part of the same bi-directional relationship

Table 3.4: Taxonomy of Basic Schema Evolution Primitives. The full algebraic expressions for these are given in Table 3.7.

can after a finite number of applications reduce any given type lattice T to a new type lattice T' which only has one type, i.e., its root.

Now assume that we have relationships in the type lattice. A repeated

application of `op9` and `op7` will after a finite number reduce the type lattice `T` to a type lattice `T'` with no relationships. The reduction of `T'` to a type lattice `T''` with a single root follows from the discussion above.

Lemma 2 *There is a finite sequence of operations* {`op1`, `op3`, `op4`, `op6`, `op8` } *that generates any desired type lattice* `T` *from an empty type lattice* `T'`*.*

Proof: Consider that we have two type lattices `T` and `T'`. Let `T` be a type lattice with a finite number of types, edges and one root, and `T'` be a type lattice with only a root node, i.e., a schema with just a root. The following procedure can, via the finite sequence of operations listed above, transform the type lattice `T'` to become identical to the type lattice `T`. Traverse type lattice `T` in a breadth-first order and perform the following for each type `t` visited:

- For every type `t` in `T`, add a corresponding type `t'` to type lattice `T'` (and hence also $\mathcal{T}$' using the operation `op1`.
- Add all local properties `p` of type `t` to the new type `t'` in `T'` using the operation `op5`.
- For the incoming edge into the type `t`, add a corresponding incoming edge to the node `t'` in `T` using the operation `op3`.

Once all the types are built, build the relationships between the types, if any, using the operation `op6` and `op8`.

The resultant type lattice `T'` is equivalent to the initial type lattice `T` as they have the same set of types and the same type lattice. □

Theorem 1 *Given two arbitrary type lattices* `T` *and* `T'`*, there is a finite sequence* `F` *of operations from* { `op1`, `op3`, `op4`, `op5`, `op6`, `op7`, `op8`, `op9` }*, such that when* `F` *is applied to the type lattice* `T` *it produces the type lattice* `T'`*.*

Proof: We can prove this by first reducing the type lattice `T` to an intermediate type lattice `T1` using Lemma 1. The type lattice `T1` can then be converted to the type lattice `T'` using Lemma 2. □

The set of operations { `op1`, `op3`, `op4`, `op5`, `op6`, `op7`, `op8`, `op9` } is a subset of the taxonomy of operations given in Table 3.4. Hence the completeness of this set of operations follows from Theorem 1.

Minimality of Evolution Primitives. Following the Orion schema evolution taxonomy [BKKK87], we have kept the schema changes *add-class*, *add-attribute*, *delete-attribute* and *add-ISA-edge* as primitives in our basic set. We have excluded the schema change *change-name-of-attribute* and *rename-class* as they can be achieved by the composition of two other primitives. For example, *change-name-of-attribute* can be accomplished by *add-attribute* followed by copying values from the old attribute to the new one followed by *delete-attribute* of the old attribute. A library of these basic transformations which are not *minimal* yet may be very common and hence useful can be provided by SERF[2]. We have replaced the schema change *drop-class* from the Orion taxonomy with the *delete-class* operation which removes a leaf class. This is due to the fact that the *drop-class* can have many different semantics, each of which can be achieved by combining other primitives

[2] In the later sections we show how SERF can be used to achieve transformations by combining the primitives given in Table 3.4.

given in Table 3.4. We also do not consider the operation *delete-ISA-edge* as part of the primitive set. The semantics of this operation can be achieved by a combination of the *delete-class* and *add-ISA-edge* operations.

A proof for the minimality of our operation set follows from Theorem 1.

`add-class` Primitive	`delete-class` Primitive

function *add-class* (*type* : t**, *types:*** $\mathcal{T}$**)**
{
pre-condition:
$t \notin \mathcal{T}$
 	
if $\mathcal{T} = \emptyset$, then *root* = t
$\mathcal{T} \leftarrow \mathcal{T} \bigcup t$
$C(root) \leftarrow C(root) \bigcup t$
 	
$P(t) \leftarrow P(t) \bigcup t$
$C(t) \leftarrow \emptyset$
$N(t) \leftarrow \emptyset$
$H(t) \leftarrow N(root)$
$E(t) \leftarrow \emptyset$
 	
}
	function *delete-class* (*type* : t**, *types:*** $\mathcal{T}$**)**
	{
	pre-condition:
	$t \in \mathcal{T}$ and
	$C(t) = \emptyset$ and
	$t \neq root$
	
	$\mathcal{T} \leftarrow \mathcal{T} - t$
	for all $s \in P_e(t)$:
	$C(s) \leftarrow C(s) - t$
	$E_e(s) \leftarrow E_e(s) - E(t)$
	delete $N(t)$
	delete $H(t)$
	delete $E(t)$
	
	}

Table 3.5: The Schema Evolution Primitives.

Consistency of Evolution Primitives. Tables 3.5 through 3.7 define the evolution primitives given in Table 3.4. No object transformations are indicated in these functions. Each of these primitives preserves the invariants listed in Section 3.2.1 and hence always produces a consistent schema when given a consistent schema. For example, the
`delete-class` primitive deletes the specified type from the set of types $\mathcal{T}$. We assume here that the schema is consistent before the operation is invoked. The `delete-class` preserves:

- Invariant 1 - by disallowing the deletion of the `root` type (precondi-

add-ISA-edge Primitive

function ***add-ISA-edge (type :*** x, ***type:*** y***)***
{
  pre-condition:
    $P(y) = \{root\}$ and
    $x \neq root$

  $P(y) \leftarrow P(y) - root$
  $C(root) \leftarrow C(root) - y$

  $C_e(x) \leftarrow C_e(x) \bigcup C_e(y)$
  $P(y) \leftarrow \{x\}$
  for all $t \in C_e(y)$:
    $P_e(t) \leftarrow P_e(t) \bigcup P_e(x)$
  $H(y) \leftarrow H(y) \bigcup N(x)$
  $H_e(y) \leftarrow H(y) \bigcup H_e(x)$

  for all $t \in C_e(y)$:
    $H_e(t) \leftarrow H_e(t) \bigcup H_e(y)$

  for all $s \in P_e(y)$:
    $E_e(s) \leftarrow E_e(s) \bigcup E_e(y)$
}

Table 3.6: The Schema Evolution Primitives.

add-attribute Primitive

function ***add-attribute (type :*** t, ***attribute:*** a***)***
{
  pre-condition:
    $a \notin N(t)$ and
    $a \notin H(t)$ and
    for all $c \in C_e(t)$:
      $a \notin N(c) \bigcup H(c)$

  $N(t) \leftarrow N(t) \bigcup a$
  for all $c \in C_e(t)$:
    $H(c) \leftarrow H(c) \bigcup a$
}

delete-attribute Primitive

function ***delete-attribute (type :*** t, ***attribute:*** a***)***
{
  pre-condition:
    $a \in N(t)$

  $N(t) \leftarrow N(t) - a$
  for all $c \in C_e(t)$:
    $H(c) \leftarrow H(c) - a$
}

Table 3.7: The Schema Evolution Primitives.

tion 3), we guarantee that this root will stay unique in the schema.

- Invariant 2 - by allowing a type to be deleted only if it is a *leaf* type,

`add-reference-attribute` Primitive

function add-reference-attribute (C_s**, r,** C_d**, default)**

{ **precondition:**

$C_s, C_d \in \mathcal{C}$ and
$\mathbf{r} \notin N(C_s)$
$N(C_s) \leftarrow N(C_s) \bigcup \mathbf{r}$
$\textit{in-path}(C_d) \leftarrow \textit{in-path}(C_d) \bigcup <C_s,\mathbf{r}>$
$\textit{out-path}(C_s) \leftarrow \textit{out-path}(C_s) \bigcup <C_d,\mathbf{r}>$
$\forall \mathbf{C}_x \in \textit{sub}^*(C_s)$
$\textit{out-path}(\mathbf{C}_x) \leftarrow \textit{out-path}(\mathbf{C}_x) \bigcup <C_d,\mathbf{r}>$

}

`delete-refernce-attribute` Primitive

function delete-reference-attribute (C_s**, r)**

{

precondition:
$C_s \in \mathcal{C}$ and
$\mathbf{r} \in N(C_s)$ and
$\textit{domain}(\mathbf{r}) \in \mathcal{C}$
$N(C_s) \leftarrow N(C_s) - \mathbf{r}$
$\textit{in-path}(\textit{domain}(\mathbf{r})) \leftarrow \textit{in-path}(\textit{domain}(\mathbf{r})) - <C_s,\mathbf{r}>$
$\textit{out-path}(C_s) \leftarrow \textit{out-path}(C_s) - <\textit{domain}(\mathbf{r}),\mathbf{r}>$
$\forall \mathbf{C}_x \in \textit{sub}^*(C_s)$
$\textit{out-path}(\mathbf{C}_x) \leftarrow \textit{out-path}(\mathbf{C}_x) - <\textit{domain}(\mathbf{r}),\mathbf{r}>$

}

`form-relationship` Primitive

function form-relationship (C_s**,** $\mathbf{r}_s$**,** C_d**,** $\mathbf{r}_d$ **)**

{

precondition:
$C_s, C_d \in \mathcal{C}$ and
$r_s \in N(C_s)$ and
$r_d \in N(C_d)$ and
$<C_s, r_s> \in \textit{in-path}(C_d)$ and
$<C_d, r_d> \in \textit{in-path}(C_s)$

$\alpha^{-1}(r_d) = r_s$ and $\alpha(r_s) = r_d$

}

`drop-relationship` Primitive

function drop-relationship (C_s**,** $\mathbf{r}_s$**,** C_d**,** $\mathbf{r}_d$ **)**

{

precondition:
$C_s, C_d \in \mathcal{C}$ and
$r_s \in N(C_s)$ and
$r_d \in N(C_d)$ and
$<C_s, r_s> \in \textit{in-path}(C_d)$ and
$<C_d, r_d> \in \textit{in-path}(C_s)$ and
$\alpha^{-1}(r_d) = r_s$ and $\alpha(r_s) = r_d$

$\neg(\alpha^{-1}(r_d) = r_s$ and $\alpha(r_s) = r_d)$

}

Table 3.8: The Schema Evolution Primitives. Here α represents the semantics of a binary relationship [CRH01].

thereby ensuring that there are no subtypes whose supertype is not in $\mathcal{T}$.

- Invariant 3 - esnures that `t` is a leaf node when deleted.

- Invariant 4 - not affected.
- Invariant 5 - not affected.

Similarly, by inspection of all the primitives given in Table 3.5 through to 3.7, we can show that they are sound and generate a consistent schema, preserving all invariants.

3.3 Summary

In this chapter we have presented the essential background that is needed for the remainder of this part on SERF. In particular we have described the ODMG object model and presented a detailed taxonomy for evolving an ODMG-compliant schema.

Chapter 4

The SERF Framework

In this chapter we present the fundamental principles of our SERF transformation framework [CJR98c, CJR98a, CRed, RCL$^+$99][1]. In particular, we demonstrate how our framework succeeds in giving users the *flexibility* to define the semantics of their choice, the *extensibility* of defining new complex transformations, and the *re-usability* of these transformations through the notion of templates.

4.1 Features of the Framework

The SERF framework addresses the limitations of current OODB technology that restrict schema evolution to a *predefined* set of operations with *fixed* semantics. In particular, our goal is to support *arbitrary user-customized* and possibly *complex* schema evolution operations. Similar to [Bré96], our first step in this direction is to allow users to build new schema transformations

[1] This work was done in collaboration with Jin Jing, a Masters student at WPI [Jin98].

with customized semantics using a fixed set of schema evolution primitives provided by the underlying OODB system (see Section 3.2.2.).

While our work is based on describing complex schema transformations using a fixed set of basic schema evolution primitives, a pure sequence of these schema evolution primitives is not always powerful enough to express many desired transformations. For example, Figure 4.1 shows a schema transformation that creates a new class `MergePapers` based on the two input classes `Author` and `Paper`. The class `MergePapers` is constructed by collecting some of the attributes that exist `Author` and `Paper`. For example, we might only want to add the attribute `AuthorName` if it exists in both the classes `Author` and `Paper`. And in this case, we want to add it only once. There is no pure sequence of schema evolution operations that could characterize this logic of attribute selection for `MergePapers`. Thus, we recognize the need to have some language as "glue logic". Moreover, such a language is also needed to achieve value-based transformations of objects across types. For example, if we are adding `AuthorName` from `Author` we might also want to take the value of `AuthorName` from the instances of `Author`. In current OODB systems, to achieve these types of transformations users have to resort to writing *ad hoc* code using a programming language. This has the drawback of being both programming language and system dependent, and hence not portable. Moreover, it is much harder task to reason and prove the consistency of a transformation written in a programming language.

We therefore advocate the use of a declarative query language, such as OQL, as a transformation language. The query language must have an in-

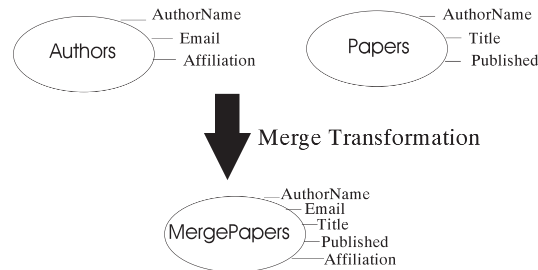


Figure 4.1: An Example Schema Graph for a MERGE Transformation.

terface for invoking the schema evolution primitives, as those provide the basic mechanism for changing the type structure. The query language must also have the expressive power for realizing any arbitrary object manipulations to transform objects from one object type to another. In Section 5.1, we will show that this approach also guarantees consistency for all schema transformations.

In our framework, these arbitrarily complex transformations can be encapsulated and generalized by assigning a name and a set of parameters to them. From here on these are called *transformation templates* or *templates* for short. By parameterizing the variables involved in a transformation, such as the input and output classes and their properties, a transformation becomes a *generalized reusable* module applicable to any application schema. By assigning a name to such a template, it can also now be *re-used* from within other transformations. This leads us to the idea of collecting templates in a *template library* and thus guaranteeing the availability of templates to any user at any time, just as the fixed set of schema evolution operations are available to users in any regular schema evolution system.

4.2 Schema Transformations

A schema transformation can be used to express different semantics for primitives as well as to create new possibly complex schema evolution operations.

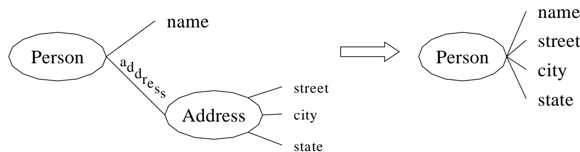


Figure 4.2: Example of the Inline Transformation.

```
// Add the required attributes to the Person class        }
add-attribute (Person, Street, String, " " );               }
add-attribute (Person, City, String, " " );                 } Step A
add-attribute (Person, State, String, " " );                }

// Get all the objects for the Person class                 }
define extents() as                                         }
    select c                                                } Step B
    from Person c;                                          }

// Update all the objects                                   }
for all obj in extents():                                   }
    obj.set (obj.Street, valueOf(obj.address.Street));      } Step C
    obj.set (obj.City, valueOf(obj.address.City));          }
    obj.set (obj.State, valueOf(obj.address.State));        }

// Delete the address attribute                             }
delete-attribute (Person, address);                         }
                                                            } Step A
// Delete the Address class                                 }
delete-class (Address);                                     }
```

Figure 4.3: SERF Representation of the Inline Transformation using OQL.

We illustrate the steps involved in a schema evolution transformation

using the example of *Inline* (also called *SLICE*) which is defined as the replacement of a referenced type with its type definition [Ler00, SHT+77]. For example in Figure 4.2 the `Address` type is inlined into the `Person` class. For this, all the attributes defined for the `Address` type (the referenced type) are now added to the `Person` type resulting in a more complex `Person` class. Figure 4.3 shows the *Inline* transformation expressed in our framework using OQL, schema modification primitives, and system-defined update methods. In this example, the `obj.set()` methods are the system-provided update methods.

In general a transformation has three types of operations where each type of operation can be composed of, or inter-mingled, with the other types of operations. Here for ease of readability, we denote each type of operation as a step. The three key steps denote the primary functionality of the step.

- **Step A: Change the Schema.** All structural changes, i.e., changes to the schema, are made through the schema evolution primitives as described in Section 3.2.2. For example, **Step A** in Figure 4.3 shows the addition of the attributes `street, city` and `state` via the *add-attribute* schema evolution (SE) primitive to the `Person` class.
- **Step B: Query the Objects.** As a preliminary to performing object transformations, we need to obtain the handles for objects involved in the transformation process. These may be objects from which we copy object values (e.g., `Address` objects in **Step B**), or objects that themselves are modified (e.g., `Person` objects in **Step C**).

- **Step C: Change the Objects.** The next step to any schema transformation is the transformation of the objects to conform to the new schema. Through **Step B**, we already have a handle to the affected object set. **Step C** in Figure 4.3 shows how a query language, such as OQL, and system-defined update methods, such as `obj.set`(...), can be used to perform object transformations.

In general, a transformation in SERF uses a query language to query over the application objects, as in **Step B**. The transformation also uses the query language to invoke the schema evolution primitives (such as `add-attribute()` in **Step A**) and update methods for updating the objects (such as `obj.set()` in **Step C**).

In this example, we simply added attribute values to the existing objects. In some transformations, existing objects might need to be deleted or new objects might be created. OQL allows for the creation of new objects through a constructor-like statement. Deletion of objects has to be done by a system-defined update method.

4.3 Transformation Templates

Figure 4.3 showed how a schema transformation that inlined the `Address` class into the `Person` class using OQL, schema primitives, and update methods. Embedding schema evolution operations in OQL allows the user to achieve any desired transformation. This is a big step above the *fixed* set of schema evolution primitives offered by previous approaches. However, this can also be compared to the functionality of a custom program in a

high-level language. We provide greater re-use for SERF transformations by now introducing the notion of *templates* [CJR98c].

A SERF template is a named, parameterized and generalized transformation. Input parameters for the template are *typed* to provide us with type-checking and hence a measure of syntactic consistency checking for SERF templates. A list of the parameter types appears in [CRH00b]. We next explain how a transformation is generalized.

4.3.1 Generalization of a Transformation

A template is a generic transformation, i.e., it should be applicable for any input. To achieve this generality (1) queries in the transformation that are specific to a particular input must be modified to now utilize variables (refer Figure 4.5); and (2) any direct schema information that is utilized in the transformation must now be procured by querying the meta-data based on the input that is provided. Thus, to achieve a SERF template, we assume that the system dictionary for the database is accessible via the query language.

Figure 4.4 shows how the meta-data can be used to achieve a generalized *inline* transformation that can now be applied for any set of parameters. Here we use the system dictionary not only to discover the class being referred to by the reference attribute `refAttrName` but also then to get the set of attributes that belong to the reference class. This *inline* template when instantiated with the variables `Person` and `address` will achieve the same effect as the *inline* transformation shown in Figure 4.3.

```
begin template inline ( Class: className, Attribute: refAttrName)
{

  refClass = element (
                select a.attrType
                from MetaAttribute a
                where a.attrName = $refAttrName
                and   a.classDefinedIn = $className; )

   define localAttrs(cName) as
      select c.localAttrList
      from MetaClass c
      where c.metaClassName = cName;

   // get all attributes in refAttrName and add to className
   for all attrs in localAttrs(refClass):
      add-attribute ($className, attrs.attrName,
                     attrs.attrType, attrs.attrDefault );

   // get all the extent
   define extents(cName) as
      select c
      from cName c;

   // set: className.Attr = className.refAttrName.Attr
   for all obj in extents($className):
      for all Attr in localAttrs(refClass):
         obj.set (obj.Attr, valueOf(obj.refAttrName.Attr));

  delete-attribute($className, $refAttrName);
}

end template
```

Figure 4.4: Generalized Inline Template based on the Inline Transformation.

4.3.2 Template Library

While most databases today provide some support for schema evolution, it is very specific to the underlying OODB. Thus, we propose the development of a library of schema evolution templates that can be ported across OODBs as long as the system requirements as set forth in [CJR98d] are met.

Thus SERF templates for a particular domain are collected in a Template Library. Simple key-word search is available on all stored parameters of the template such as the input and output parameters, the name, and its description. *Inheritance* and other semantic relationships between tem-

plates are defined to facilitate a multi-level organization of the template library. Such a template library will be an important resource in the schema evolution community, much like other re-usable libraries in software engineering.

4.3.3 SERF Template Language

A SERF template is thus a named sequence of OQL statements extended with a name, parameters and variables that can be translated to pure OQL statements during the process of instantiation. The `BNF` for a SERF template is given in Figure 4.5. In the BNF `restricted_query` denotes the OQL query limited to invoking only object updates and the schema evolution primitives.

4.3.4 SERF Template Instantiation and Processing

During the instantiation process, the SERF template specification is translated to pure OQL statements. Prior to the instantiation it is assumed that a database on which the template is to be executed has already been selected, and we are in the context of this database environment. A consistency and bind check is performed both before and after the instantiation of the template. Figure 4.6 shows the steps for the execution of a template. When a template is created, the user can assign a name to it and also specify its parameters. A compile-time syntax check is performed for both the OQL queries and the $system_function$ in the template. When the template is instantiated, the caller provides bindings for the parameters. A second

$$
\begin{array}{rcl}
template & ::= & \text{begin} \quad \text{template} \\
template_name & & ([paramList]) \\
 & & template_statements \\
 & & \text{end} \quad \text{template} \\
template_statements & ::= & template_statement; \quad | \\
 & & template_statement; template_statements \\
template_statement & ::= & \textbf{\textit{define_query}} \quad | \quad query \\
\textbf{\textit{define_query}} & ::= & \textbf{define} \quad \textbf{\textit{identifier}} \quad ([paramList])\text{as} \quad query \\
query & ::= & oqlquery \quad | \quad restricted_query \\
restricted_query & ::= & oqlquery([function]^*) \quad | \quad \lambda \\
function & ::= & system_function(basic_query^*) \quad | \\
 & & schema_primitive(parameter^*) \\
paramList & ::= & parameter \quad | \quad parameter, paramList \\
parameter & ::= & type \quad variable \quad | \\
 & & string_literal \\
type & ::= & Class \quad | \quad Attribute \quad | \quad MetaClass \quad | \quad MetaAttribute| \\
 & & string_literal \\
variable & ::= & string_literal \\
basic_query & ::= & nil \quad | \quad true \quad | \quad false \quad | \quad literal \\
oqlquery & ::= & oql \quad bnf(omittedhere)
\end{array}
$$

Figure 4.5: The BNF for a SERF Template.

check performs type-checking, i.e., ensures the parameters are of the type specified for the template, and validates the bindings, i.e., ensures that all parameters (except those whose type is a base type such as `String`, `int` or `boolean`) exist in the schema repository within the scope specified. For example, if `MetaClass Person` is an input parameter, then `Person` should exist in the schema repository under the scope of the `Schema`.

A successful check leads to the actual execution of the template. Each template is atomic and is thus executed in one transaction. The regular

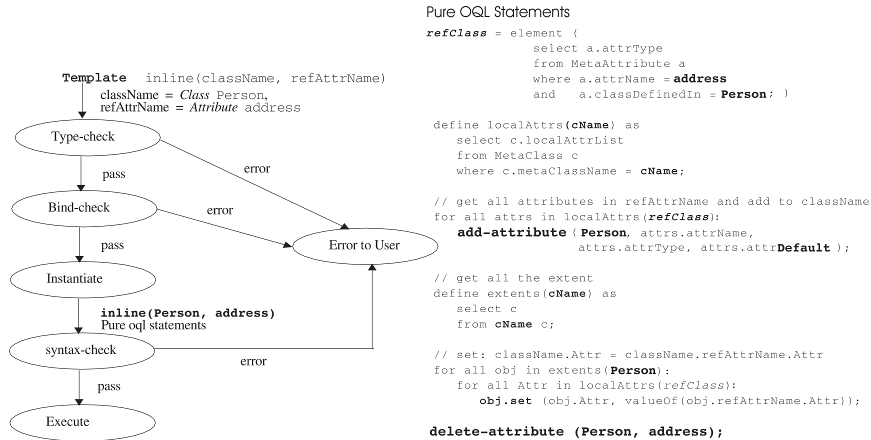


Figure 4.6: Steps for the Execution of a Template

transaction semantics apply for this. The template which is now a sequence of OQL statements is executed using the OODB's OQL Query Engine. Each primitive embedded in the OQL statements is executed against the database and can be executed in a nested transaction if needed.

4.4 Summary

In this chapter we have introduced the basic concepts of the SERF framework. This is a core contribution for the SERF work and has resulted in several publications [CJR98c, CRed, CNR00, CRH01, CJR98a]. It should be noted here that the number of transformations that can be done using SERF are directly proportional to the taxonomy of schema evolution operations provided by the underlying system.

Chapter 5

Soundness and Consistency of SERF Templates

In this chapter, we show how the SERF templates can assure the soundness of the database after their execution as well as show how users can specify their own constraints which must then be preserved by the execution of the template [CRed, CRH00a, CRH01][1].

5.1 Soundness of Templates

An important property imposed on schema operations is that their execution on a consistent schema always results in a *consistent* new schema [BKKK87] with consistency as defined by the invariants of the data model. In Section 3.2.2 we have showed via an example and by complete defini-

[1]This work was done in collaboration with George T. Heineman, Associate Professor at WPI.

tion of the primitives, that the schema change primitives for the ODMG object model as defined in Section 3.1 indeed preserve the consistency of a schema. Thus, an individual schema evolution primitive (Section 3.2.2) when applied on a consistent schema will always generate a consistent schema. In this section, we now extend this notion of soundness to SERF templates.

We utilize the type lattice and the set of nine operations on the type lattice, represented by $\texttt{OP}$, presented in Section 3.2.2 to show the soundness of SERF templates. Here the type lattice $\texttt{T}$ represents a schema conforming to the invariants and the operations correspond to the evolution primitives in Table 3.4. Towards showing soundness of SERF templates, we first state the following lemma.

Lemma 3 *A sequence* $\texttt{P}_n$ *of operations* $\langle \texttt{o}_1, \ \dots \ o_i \ \dots \ \texttt{o}_n \rangle$*, where* $\texttt{o}_i \in \texttt{OP}$ *(Table 3.4), and* $\texttt{1} \leq \texttt{i} \leq \texttt{n}$ *when applied to a consistent type lattice* $\texttt{T}$ *generates a consistent type lattice* $\texttt{T'}$*.*

Proof: By Induction on number of operations $\texttt{op}_i$ in a sequence:

Basis - Let $\texttt{P}$ be a sequence of one operation, i.e., $\texttt{n} = 1$.

$$\texttt{P}_1 = \langle \ \texttt{o}_1 \ \rangle, \text{ where } \texttt{o}_1 \in \texttt{OP}.$$

We have shown in Section 3.2.2 that any operation $\texttt{o} \in \texttt{OP}$ when applied to a consistent type lattice $\texttt{T}$ generates a consistent type lattice $\texttt{T'}$ or leaves the original lattice unchanged. This is denoted as $\texttt{T} \overset{\texttt{P}_1}{\Longrightarrow} \texttt{T'}$.

Assumption: For a sequence $\texttt{P}_k$ of k-operations, with $\texttt{P}_k = \langle \texttt{o}_1 \dots \texttt{o}_i \dots \texttt{o}_k \rangle$, where $\texttt{o}_i \in \texttt{OP}$, $\texttt{i} = \texttt{1} \dots \texttt{k}$ and $\texttt{T}$ is a consistent type lattice, then $\texttt{T} \overset{\texttt{P}_k}{\Longrightarrow} \texttt{T'}$ generates a consistent type lattice $\texttt{T'}$.

Induction: Now let $\mathtt{n} = \mathtt{k+1}$ such that $\mathtt{P}_{k+1}$ is a sequence of $\mathtt{k + 1}$ operations and $\mathtt{P}_{k+1} = \langle\ \mathtt{o}_1 \ldots \mathtt{o}_i \ldots \mathtt{o}_k,\ \mathtt{o}_{k+1}\ \rangle$, where $\mathtt{o}_i \in \mathtt{OP}$, $\mathtt{i} = \mathtt{1} \ldots \mathtt{k+1}$. That is, $\mathtt{P}_{k+1} = \mathtt{P}_k \bullet \mathtt{o}_{k+1}$. We need to prove the following:

$$\mathtt{T} \overset{\mathrm{P_{k+1}}}{\Longrightarrow} \mathtt{T'}.$$

To achieve this we can apply the first $\mathtt{k}$ operations of $\mathtt{P}_{k+1}$. Let us denote them by $\mathtt{P}_k$. We know from our assumption that $\mathtt{P}_k$, when applied to a consistent type lattice $\mathtt{T}$, produces another consistent type lattice $\mathtt{T}_k$. We can then apply the $\mathtt{k+1}^{th}$ operation to $\mathtt{T}_k$ by induction basis. Thus we have,

$$\mathtt{T} \overset{\mathrm{P_k}}{\Longrightarrow} \mathtt{T}_k \overset{\mathrm{op_{k+1}}}{\Longrightarrow} \mathtt{T'},$$

Thus we have, $\mathtt{T} \overset{\mathrm{P_{k+1}}}{\Longrightarrow} \mathtt{T'}$ and the generated $\mathtt{T'}$ is a consistent type lattice. □

Using the above result, we now state the theorem for the soundness of the SERF Templates.

Theorem 2 *A SERF template when applied to a consistent schema conforming to the specified object model (Section 3.1) will always result in a consistent schema as defined by the invariants in Section 3.2.1.*

Proof: A SERF template has three main components: (1) schema evolution primitives; (2) queries on the application or meta-data; and (3) object updates expressed in OQL. We exclude (2) and (3) on the basis that these do not in any way alter the structure of the schema and hence do not violate the invariants of the object model as presented in Section 3.2.1. In a SERF template any composition of schema evolution primitives results in the sequential execution of the primitives in that composition. The soundness of the SERF templates follows from Lemma 3. □

Here we do not guarantee the consistency of the data after the application of a SERF template, but only that of the schema. Data updates are made during the execution of a schema evolution primitive and by the queries in the SERF template. Guaranteeing the consistency of these data updates is outside the scope of discussion here.

5.2 Contracts

Conventionally, schema evolution primitives contain hard-coded constraints that parallel the invariants of the object model. These constraints must be satisfied in order to guarantee the consistency of the system. While a SERF template also preserves these invariants as shown in Section 5.1, the SERF template may itself have additional constraints that now specify user-level constraints that are to be imposed on the SERF template. We introduce the notion of *contracts*, a declarative mechanism for expressing the constraints for a template and now define a ROVER Wrapper as a template with contracts. This approach also has the advantage that a change in the object model results in a change in the invariants of the model and consequently in the re-engineering of the schema evolution primitives. Contracts can now also be used to decouple the invariant constraints from the actual implementation of the schema evolution primitives. Changes to the invariants of the object model now merely result in the update of the declarative contracts associated with the evolution operations rather than the update of the actual system code. Below we briefly introduce *contracts* and show how the decoupling of constraints can be achieved.

Contracts provide a declarative description of the behavior of a template (or primitive) as well as a mechanism for expressing the constraints that must be satisfied prior to the execution of the actual evolution primitive. Contracts are divided into two categories **preconditions** and **postconditions**.

The constraints, termed *preconditions*, are placed prior to any body of template code (OQL statements including system-defined schema evolution primitive). The *preconditions* are separated from the actual OQL statements by means of the keyword **requires**. *Postconditions*, a set of contracts that appear after the body of the actual schema evolution operation at the end of the SERF template, specify the behavior of the primitives. These postconditions are preceded by the keyword **ensures** and describe the exact changes that are made to the schema by the evolution operator and hence its behavior.

Example: As an example consider the addition of relationship constructs to the object model of an OODB system. An upgrade to the schema evolution facility in this case requires new schema evolution primitives to handle the creation, modification, and deletion of uni-directional and/or bi-directional relationships. This upgrade cannot be circumvented and hence new schema evolution primitives must be added to the system. However, an update of all existing schema evolution operations to conform to the new set of invariants is also required. For example, to delete a class prior to the existence of relationships, the constraint that a class needed to be a *leaf* class was necessary to ensure that the resulting schema and database was

consistent, i.e., `delete-class` preserved the database consistency. With the addition of relationships, this constraint alone is not sufficient. We now also need to ensure that the `to-be-deleted` C_s class is not referred to by another class. Moreover, no objects in the database must refer to the objects of the C_s class. So while the conditions that need to be enforced prior to the execution the schema evolution operation have to be upgraded, the actual actions of the operations do not change. Hence the evolution primitive `delete-class` itself does not change.

Figure 5.1 shows the constraints after the addition of a relationship for the `delete-class` primitive as preconditions[2]. Thus, in this model it is easy to extend or modify the constraints without re-writing the code for the evolution primitive. Figure 5.2 shows the constraints that must be satisfied after the execution of the template code.

delete-class (C_s **)**
{
 requires:
  $C_s \in \mathcal{T} \wedge$
  $\mathcal{C}(C_s) = 0 \wedge$
  $\textit{in-degree}(C_s) = 0 \wedge$
  $\forall\, o_i \in \textbf{extent}(C_s)$
   $\textit{obj-in-degree}(o_i) = 0$

 `template body here`
}

Figure 5.1: Preconditions for Delete-Class Primitive in Contractual Form.

delete-class (C_s **)**
{ `template body here`

 ensures:
  $C_s \notin \mathcal{C} \wedge$
  $\forall\, <C_x, r_x> \in \textit{out-paths}(C_s)$
   $(<C_s> \notin \textit{in-paths}(C_x))$
$\wedge$
  $\forall\, C_x \in P_e(C_s)$
   $(C_s \notin \textit{sub}(C_x))$
}

Figure 5.2: Postconditions for the Delete-Class Primitive Template. We assume here that the meta-dictionary information such as $P_e(C_x)$ are available until the template is done executing.

[2]The notation used here is a set-theoretic version of the contract language.

Figure 5.3 depicts the inline template with contracts to specify additional pre conditions and post conditions (beyond the invariant constraints for the schema evolution primitives) for the template. The pre condition $C_s \neq \texttt{domain}(r_s)$ specifies that the reference attribute r_s is not a self-referential attribute.

begin template inline (C_s, r_s **)**
{

requires:

$C_s \in \mathcal{C} \wedge$
$\sigma(C_s) \in \textbf{types}(\mathcal{C}) \wedge$
$r_s \in N(C_s) \wedge$
$\text{domain}(r_s) \in \mathcal{C} \wedge$
$C_s \neq \text{domain}(r_s)$

ensures:

$C_s \in \mathcal{C} \wedge$
$\sigma(C_s) \in \textbf{types}(\mathcal{C}) \wedge$
$r_s \notin N(C_s) \wedge$
$\text{domain}(r_s) \notin \mathcal{C} \wedge$
$\forall\, a_x \in N(\text{domain}(r_s))$
$(a_x \in N(C_s)$

}

Body of inline template

Figure 5.3: Inline ROVER Wrapper with Set-Theoretic Contracts.

5.3 Verification of ROVER Wrappers

There are many different approaches to verify the correctness of the ROVER Wrappers. Perhaps, the most straightforward approach is that of checking the preconditions prior and postconditions after the execution of the ROVER Wrapper. If the preconditions are initially satisfied and the postconditions are met after the execution of the template, then the ROVER Wrapper is said to be correct. As an example, consider the *inline* example given in Figure 4.4. The inline template inlines all the attributes (and hence all of the data) of the class referred to by the reference attribute `refClass`

into the class className. Once all attributes and data have been moved over to className, the refClass is deleted. Now consider the scenario that the reference attribute is a self-referencing attribute[3]. In such a case after the execution of the delete-class primitive, the class refClass = className is deleted which is clearly not the desired result as indicated by the postconditions that require that className must still exist on completion of ROVER Wrapper execution. All changes made by the *inline* template must be rolled back in such a scenario.

Thus while this approach verifies the correctness of the ROVER Wrapper, its fall-back for an erroneous change is a transaction roll-back which in this case would be a rather expensive operation and hence not desirable. An alternative to this is to use theorem proving to verify the correctness of the ROVER Wrappers prior to their execution. A theorem prover based on facts postulates the different states of the system for every statement that is executed. Consider that the system is in state S_0 of the system in which all preconditions given in Figure 5.3 for the *inline* wrapper are met. The theorem prover can now calculate the new state S_1 which results from the addition of an attribute. Let state S_n represent the state of the system after all attributes have been added. In state S_n, a delete-class primitive would take the system to its final state S_f. If the state S_f does not meet the desired postconditions as stated in the wrapper, then the wrapper is not allowed to execute. In the following sections, we detail the basic requirements for a theorem prover and show via an example (*inline* template) how the postconditions of the wrapper can be checked prior to execution.

[3] Inline assumes only one level of inlining, so recursion is not an issue here.

5.3.1 Theorem Prover for SERF Templates

Verification of any program relies on the knowledge that given a start state, the program code will take the system to a desired final state. Thus, before verification can be applied, it becomes essential to describe these states, the start and the final states. In this section, we now first define these states and then walk-through an example to show how the technique can be used to verify a ROVER Wrapper.

Theorem proving approaches verification by formalizing (1) a model of computation, (2) the specification and (3) the rules of inference [BHJ+96]. The axioms and other knowledge about the environment comprise the model of computation, the pre and postconditions are the specification, i.e., the initial state of the system as well as the final targeted state that must be reached. The rules of inference are the functions that help reason about the validity of the path from the initial to the final states of the system. Table 5.1 shows what these components correspond to in the context of SERF. In the following subsections, we detail these three components for our problem domain to show via a detailed example the viability of theorem provers for SERF templates.

Theorem Prover Component	SERF Components
Model of Computation	Object Model, Invariants, System Functions
Specification	ROVER Wrappers (Pre and Post Conditions)
Rules of Inference	Schema Evolution Primitives

Table 5.1: Theorem Prover Components for the SERF Environment.

Model of Computation for ROVER Wrappers. The model of computation formally describes the environment in which the theorem prover is being applied. The SERF framework is based on the ODMG object model (Section 3) [Cea97]. Hence, the theorem prover must be provided with a formal definition of the ODMG object model, its invariants and the functionality of each of the system dictionary functions as described in Table 3.3. This model of computation is part of the setup of the theorem prover system and thus would be created once a-priori for the SERF system. It would only need to be modified if and when there is a change in the environment itself, for example if the object model changes. While different theorem provers use different languages [GM93, ORS92], for the purpose of this paper, we assume the language of the theorem prover to be set-theoretic.

Specification of ROVER Wrappers. A theorem prover requires the specification of the initial state of the system as well as the final state that needs to be verified. The contracts, i.e., the pre and postconditions as defined in Section 5.2, fulfill these requirements by providing an initial state (the preconditions) that must be valid and an expected final state (the postconditions) that must be met. However, the theorem prover expects its inputs to be expressed in a formal language. Hence these contracts would need to be converted to the language of the theorem prover, i.e., in our case to a set-theoretic language.

Rules of Inference. The rules of inference are operations that move the system from one given state to another state, i.e., code segments that take

the system from an initial specification to a final specification. The body of a ROVER Wrapper, i.e., the actual schema evolution functions and OQL code, are hence the rules of inference in our system.

For example, Figure 4.4 depicts a SERF template that inlines the class `refClass` referred to by the reference attribute `refAttrName` in class `className` into the class `className`. Each step (statement, OQL, or schema evolution function) of the inline template as shown in Figure 4.4 is a rule of inference. Each of these rules is applied one at a time to a given state of the system.

In addition to the primitive evolution programs, we also consider the `for all` OQL statement. This is translated to a repetitive application of the loop body that results in a cumulative effect on the state of the system. For example, `for all x in attributeSet: add-attribute (C, x, default)` results in the application of the `add-attribute` primitive `count(attributeList)` times, where `count` gives the number of elements in a set. The final state of the system will be the cumulative result of applying all `add-attribute` primitives. Thus you can't evaluate until the `for all` loop has completed, i.e., the entire `for all` loop is a single step.

5.3.2 Formal Verification Process: Application to Inline Template

In this section, we illustrate the working of the theorem prover by a step by step verification of a template (namely the `inline` template from Figure 5.3), thereby showing how theorem provers can be applied to our domain for verification of schema evolution transformations. This is an au-

tomated process that assumes the computation model (Section 5.3.1) and the rules of inference (Section 5.3.1) have already been provided as part of the tool. The user only needs to input the specification contracts and the template code in OQL.

Consider the ROVER inline Wrapper shown in Figure 5.3 and Figure 4.4. The class `C`$_s$ and the reference attribute `r`$_s$ must exist otherwise it is meaningless to proceed with the verification. Given this initial state `S`$_0$, we proceed to apply the first evolution change in the template, `add-attribute` (Figure 4.4). This results in a new state `S`$_1$. As part of our work, we wrap each individual change primitive program using a ROVER Wrapper with their specific contracts. Henceforth we consider the ROVER Wrapper for each evolution primitive program. Thus, in this example the initial state of the system must meet the precondition for `add-attribute`. The postconditions specified by `addattribute` represent the final state S_1 and are indicative of the behavior of the `add-attribute` primitive. Figure 5.4 lists the ROVER Wrapper for `add-attribute`. Figure 5.1 and 5.2 list the ROVER Wrapper precondition and postcondition for the `delete-class` primitive which are used in the example here.

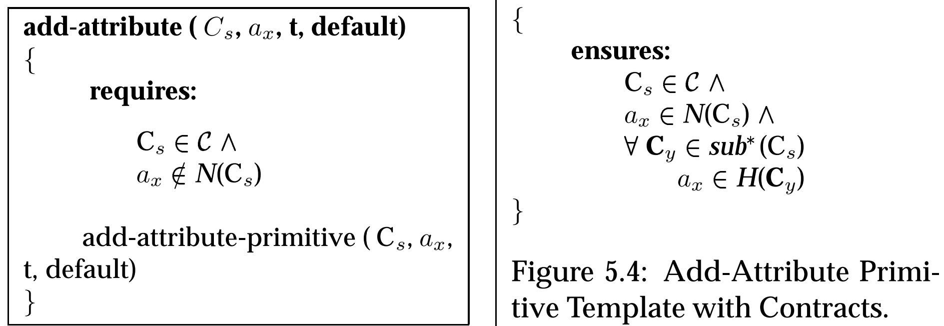


Figure 5.4: Add-Attribute Primitive Template with Contracts.

A subsequent evolution change, `add-attribute`, uses this state S_1 as its initial state against which its precondition must match. Tables 3.4 contains the schema evolution operations that we consider as rules of inference in our system. The theorem prover proves the correctness of the inline transformation shown in Figure 4.4 by first proving three theorems where each theorem is similar to the `add-attribute` and then composes them together to prove the correctness of the inline transformation itself (the fourth theorem). Each of the theorems specifies the properties of one of the evolution programs in the inline transformation.

Schema Evolution Primitive: add-attribute. The preconditions and postconditions from the contract specification in Figure 5.3 that must hold for `add-attribute(`$\mathtt{C}_s$`, `$\mathtt{a}_x$`, type, default)` are given in Equation 5.1[4]:

$$\left.\begin{array}{rcll} C_s & \in & \mathcal{C} & \wedge \\ a_x & \notin & N(C_s) & \end{array}\right\} \quad (5.1)$$

The desired postconditions expected after applying `add-attribute` are:

$$\left.\begin{array}{rcll} C_s & \in & \mathcal{C} & \wedge \\ \sigma(C_s) & \in & \text{types}(\mathcal{C}) & \wedge \\ a_x & \in & N(C_s) & \wedge \\ \forall \mathbf{y} & \in & sub^*(C_s) & \\ a_x & \in & H(\text{y}) & \end{array}\right\} \quad (5.2)$$

Theorem 3 *If the* `add-attribute` *template is applied to arguments satisfying the precondition given in Equation 5.1, then the program results satisfy the*

[4]These are repeated from Figure 5.3 for convenience.

postcondition given in Equation 5.2.

Proof: Assume that the precondition in Equation 5.1 holds and `add-attribute` adds the attribute a_x to the class C_s (Table 3.5), i.e.:

$$N(C_s) = a_x \quad \cup \quad N(C_s) \; \} \tag{5.3}$$

The primitive `add-attribute` also adds a_x to all the subclasses ***sub*** (C_s)* of C_s (refer Table 3.4). Hence we have:

$$\left.\begin{array}{rcl} \forall \mathbf{C}_y & \in & \textbf{\textit{sub}}^*(C_s) \\ H(C_y) & = & a_x \\ & \cup & H(\mathrm{C_y}) \end{array}\right\} \tag{5.4}$$

Here Equations 5.3 and 5.4 show the altered states of the system after the execution of each `add-attribute` function. From Equations 5.3 and 5.4, we have the desired postcondition as specified in Equation 5.2. □

In Figure 4.4, the `forall` loop copies all the attributes of the class $\mathtt{C}_{refClass}$[5]. At the end of the `forall` loop, with repetitive application of `add-attribute`, the desired state is given by the postcondition in Equation 5.5.

$$\left.\begin{array}{rcll} C_s & \in & \mathcal{C} & \wedge \\ \sigma(C_s) & \in & \mathrm{types}(\mathcal{C}) & \wedge \\ \forall a_k & \in & N(C_{refClass}) & \\ (a_k & \in & N(C_s)) & \wedge \\ \forall C_y & \in & \textbf{\textit{sub}}^*(C_s) & \\ (a_k & \in & H(\mathrm{C_y})) & \end{array}\right\} \tag{5.5}$$

Theorem 4 *If the* `add-attribute` *function is correct as per Theorem 3, then*

[5] $C_{refClass}$ is the class that is being referred to by the reference attribute r_x in class C_s.

repetitive execution of `add-attribute` ***for all*** $a_k \in C_{refClass}$ ***results in a cumulative effect such that postcondition given in Equation 5.5 are satisfied.***

Proof: (Proof By Induction)

Base Case: Assume that the class $C_{refClass}$ has only one attribute *a*. This reduces the `for all` statement to a simple `add-attribute` (C_s, *a*, *a.attrType*, *a.defaultValue*). We know by Theorem 3 that if the pre-condition given in Equation 5.1 holds for these arguments, then the postcondition as given in Equation 5.2 will also hold, i.e., for one attribute ($n = 1$), postcondition in Equation 5.5 reduce to postcondition in Equation 5.2.

Induction Hypothesis: Assume that the theorem holds true when class $C_{refClass}$ has k attributes and they are added to class C_s, i.e., the postcondition given in Equation 5.5 is satisfied for k `add-attribute` applications.

Induction: Prove that the post-condition in Equation 5.5 holds when the class $C_{refClass}$ has $k+1$ attributes.

We know that the postcondition (5.5) holds when class $C_{refClass}$ has k attributes and they are added to the class C_s. Assume we now have attribute a_{k+1}, the $k + 1^{th}$ attribute, that is unique, i.e., $a_{k+1} \neq a_j$, for $1 \leq j \leq k$. To add this to class C_s we do: `add-attribute`(C_s, a_{k+1}, a_{k+1}*.attrType*, a_{k+1}*.defaultValue*). We know by Theorem 3 that if this satisfies the precondition given in Equation 5.1, then the postcondition in Equation 5.2 holds true (Base Case). Combining the postcondition for the addition of k attributes (Induction Hypothesis) with the postcondition of the Base Case, we get the postcondition as given in Equation 5.5. □

Schema Evolution Primitive delete-attribute. First we give the initial state of the system, i.e., the precondition in Equation 5.6 that must be satisfied for the primitive `delete-attribute(`C_s`, `a_x`)`:

$$\left.\begin{array}{lll} C_s & \in & \mathcal{C} \quad\quad\quad \wedge \\ a_x & \in & N(C_s) \end{array}\right\} \tag{5.6}$$

After the execution of `delete-attribute`, the final state of the system is as given in the postcondition in Equation 5.7.

$$\left.\begin{array}{r} a_x \notin N(C_s) \quad\quad\quad \wedge \\ \forall \mathbf{C}_y \in \textbf{\textit{sub}}^*(C_s) \\ a_x \notin H(\mathrm{C_y}) \end{array}\right\} \tag{5.7}$$

Theorem 5 *If* `delete-attribute` ***is applied to arguments satisfying precondition (5.6), then the result satisfies the postcondition (5.7).***

Proof: Similar to Theorem 3.

Schema Evolution Primitive delete-class. The necessary precondition that must hold for `delete-class(`$C_{refClass}$`)` is given in Equation 5.8:

$$\left.\begin{array}{rcll} C_{refClass} & \in & \mathcal{C} & \wedge \\ \textbf{\textit{sub}}(C_{refClass}) & = & \emptyset & \wedge \\ \textbf{\textit{in-degree}}(C_{refClass}) & = & 0 & \wedge \\ (\forall o_i \in \textbf{extent}(C_{refClass}) : (\textbf{\textit{obj-in-degree}}(o_i) & = & 0)) & \end{array}\right\} \tag{5.8}$$

The desired postcondition (5.9) after the application of `delete-class` is as given in Equation 5.9.

$$\left.\begin{array}{rcl} \forall < C_x, r_x > & \in & \textbf{\textit{out-paths}}(C_{refClass}) \\ (< C_{refClass} > & \notin & \textbf{\textit{in-paths}}(C_x)) \quad \wedge \\ \forall C_x & \in & \textbf{\textit{super}}(C_{refClass}) \\ (C_{refClass} & \notin & \textbf{\textit{sub}}(C_x)) \quad \wedge \\ C_{refClass} & \notin & \mathcal{C} \quad \wedge \\ \sigma(C_{refClass}) & \notin & \textbf{types}(\mathcal{C}) \end{array}\right\} \tag{5.9}$$

Theorem 6 *If* `delete-class` ***is applied to arguments satisfying the precondition in Equation 5.8, then the result satisfies the postcondition in Equation 5.9.***

Proof: Similar to Theorem 3.

The Inline Transformation. To verify the correctness of the inline transformation, we chain the results of Theorems 3, 4, 5 and 6. The overall precondition for this is given by Equation 5.10[6]. The execution of the inline transformation must result in the final state as specified by Equation 5.11[7].

$$\left.\begin{array}{rcl} C_s & \in & \mathcal{C} \quad \wedge \\ a_x & \in & \textit{N}(C_s) \quad \wedge \\ C_{refClass} & = & domain(a_x) \quad \wedge \\ C_{refClass} & \in & \mathcal{C} \quad \wedge \\ C_s & \neq & C_{refClass} \end{array}\right\} \tag{5.10}$$

$$\left.\begin{array}{rcll} C_s & \in & \mathcal{C} & \wedge \\ C_{refClass} & \notin & \mathcal{C} & \wedge \\ \forall a_k & \in & \textit{N}(C_{refClass}) & \\ (a_k & \in & \textit{N}(C_s)) & \wedge \\ \forall \mathbf{C}_y & \in & \textbf{\textit{sub}}^*(C_s) & \\ (a_k & \in & \textit{H}(\mathbf{C}_y) & \wedge \\ a_k & \notin & N(C_s)) & \end{array}\right\} \tag{5.11}$$

[6]This is the precondition for the `inline` wrapper in Figure 5.3.
[7]This is the postcondition for the `inline` wrapper given in Figure 5.3.

Theorem 7 *If Theorems 3, 4, 5, and 6 are satisfied in the order specified, then the inline transformation satisfies the postcondition given in Equation 5.11.*

Proof: The proof for this can be given by a combination of the postcondition in Equations 5.5, 5.7 and 5.9. □

Using theorem proving techniques as shown for the template here it is possible to verify the correctness of any given template. If at any point one of the sub-theorems is not satisfied, i.e., if Theorem 3, 4, 5, or 6 is not satisfied, the verification process will be aborted and the template not permitted to execute. However, all of this will occur prior to any execution of the ROVER Wrapper. Hence, any failure of the verification process will result in no execution of the ROVER Wrapper. While such a verification process may incur additional overhead of checking, we believe it can be shown that under erroneous conditions the verification process can save hours when compared to a roll-back strategy.

5.4 Summary

In this chapter, we have addressed the issue of consistency of the DBMS systems after the execution of complex transformations such as a SERF transformation. We have defined two levels of consistency, *invariant preserving*, that ensures that after the execution of a SERF template, the resultant OODB database conforms to the invariants of the data model. We have shown show that that if each schema evolution operation is an invariant preserving operation, then a SERF template is also invariant preserving [CRed]. Within the context of SERF, we have identified the need to

define *template semantic* consistency that would allow users to specify consistency requirements for their templates. To enable users to specify these template-semantic constraints, we have introduced the notion of *Template Wrappers* [CRH00b]. These Template Wrappers, based on Software Contracts [Mey92], allow a user to specify semantic constraints on a template which can then be checked at runtime. Such contracts can also be checked prior to the execution of the template. We have shown via an example that theorem-provers may be a feasible option in this area [CRH01].

Chapter 6

Reducing the Runtime of SERF Templates

In this chapter we look at reducing the run-time of SERF templates. Specifically, we look at reducing the run-time of a template. For this we present the `CHOP` approach [CNR00, CNR99][1].

6.1 Optimizing the Performance of Schema Evolution Sequences

Schema evolution is an expensive process both in terms of system resource consumption as well as database unavailability [FMZ94b]. This expense of schema evolution is further emphasized by now having complex evolution operations that string multiple primitive schema evolution operations in a

[1] This work was done in collaboration with Chandrakant Natarajan, a Master's student at WPI.

sequence as in the case of SERF templates. As a first step of optimization, we present an approach that disregards the queries and instead focuses on reducing the run-time execution of just the schema evolution operations.

In previous work, researchers have looked at improving system availability during schema evolution by proposing execution strategies such as deferred execution [Tec94, FMZ94b]. No work, however, has been undertaken to actually optimize or reduce the sequence of operations that are being applied to a given schema. Kahler *et al.* [KR87] have looked at pre-execution optimization for reducing the number of update messages that are sent to maintain replicated sites in the context of distributed databases. In their approach, the messages are simple data updates on tuples. The messages are sorted by their tuple-identifier, and then the change history of the tuple is condensed (with merge or remove) into one update operation.

We now present a similar approach (merge, cancel, eliminate) for optimizing a sequence of schema evolution operations. Our approach is orthogonal to the existing execution strategies for schema evolution, i.e., it can in fact be applied to both immediate and the deferred execution strategies [FMZ94b]. The optimization strategy, called CHOP, exploits two principles of schema evolution execution within one integrated solution:

- Minimize the number of schema evolution operations in a sequence by canceling or eliminating schema evolution operations. For example, adding an attribute and then deleting the same attribute is an obvious case of cancellation where neither operation needs to be exe-

cuted.

- Merge the execution of all schema evolution changes that operate on one extent to amortize the cost of schema evolution over several schema changes. For example, consider a sequence that adds two or more attributes to the same class. Object updates for these done simultaneously can potentially reduce the cost of executing these sequentially by 50%.

In this section we present a general strategy for reducing a given sequence of schema evolution operations **prior** to its actual execution. Our work is based on a taxonomy of schema evolution operations we developed for the ODMG object model as given in Table 3.2.2 but it can easily be applied to any other object model. We present here an analysis of the schema evolution operations and the schema to characterize the conditions under which operations in a sequence can be optimized. Based on this analysis we present the *merge*, *cancel* and *eliminate* optimization functions and the conditions under which they can be applied. We have also been able to show both formally and experimentally that the order in which these functions are applied is not relevant for the final optimized sequence, i.e., they will all produce the same *unique* final sequence. As a conclusion to our work we also present a summary of our experimental results.

6.1.1 Foundations of Schema Evolution Sequence Analysis

To establish a foundation for our optimization principles we have developed a formal characterization of the schema evolution operations, their

impact on the schema, as well as their interactions within a sequence.

Term	Description	Capacity Effects
add-class(c, $\mathcal{C}$)	Add new class c to $\mathcal{C}$ in schema S (AC)	augmenting
delete-class(c)	Delete class c from $\mathcal{C}$ in schema S if *subclasses(C)* = $\emptyset$ (DC)	reducing
rename-class(c, d)	Rename class c to d (CCN)	preserving
add-ISA-edge(c_x, c_y)	Add an inheritance edge from c_x to c_y (AE)	augmenting
delete-ISA-edge(c_x, c_y)	Delete the inheritance edge from c_x to c_y (DE)	reducing
add-attribute(c_x, a_x, t, d)	Add attribute a_x of type t and default value d to class c_x (AA)	augmenting
delete-attribute(c_x, a_x)	Delete the attribute a_x from the class c_x (DA)	reducing
rename-attribute(a_x, b_x, c_x)	Rename the attribute a_x to b_x in the class c_x (CAN)	preserving

Table 6.1: The Taxonomy of Schema Evolution Primitives.

Schema evolution operations are generally categorized as **capacity-augmenting** if they increase the capacity of the schema, for instance, by adding a class, **capacity-reducing** if they decrease the capacity of the schema, for instance, by deleting a class, or **capacity-preserving** if they do not change the capacity of the schema, for instance, by changing the name of a class [RR97]. For each schema evolution primitive its capacity type is shown in the third column of Table 6.1.

Table 6.2 defines the various relationships that can exist in general between schema evolution operations. Table 6.3 applies these to the schema evolution operation presented in Table 6.1.

Operation Relation	Description
same-operation-as	`op1` is **same-operation-as** `op2` if they both have the same operation name irrespective of the particular parameters they are being applied to.
inverse-operation-of	`op1` is **inverse-operation-of** `op2` if the effects of one operation `op1` could be canceled (reversed) by the effects of the other operation `op2`.
super-operation-of	`op1` is **super-operation-of** `op2` if the functionality of `op1` supercedes the functionality of `op2`, i.e., `op1` achieves as part of its functionality also the effects of `op2`

Table 6.2: Classification of Operation Properties.

	AC()	**DC()**	**CCN()**	**AA()**	**DA()**	**CAN()**	**AE()**	**DE()**
AC()	same	inverse	-	super	-	-	-	-
DC()	inverse	same	super	super	super	super	super	super
CCN()	-	-	same inverse	-	-	-	-	-
AA()	-	-	-	same	inverse	-	-	-
DA()	-	-	-	inverse	same	super	-	-
CAN()	-	-	-	-	-	same	-	-
AE()	-	-	-	-	-	-	same	inverse
DE()	-	-	-	-	-	-	inverse	same

Table 6.3: Classification of Operation Properties for the Schema Evolution Taxonomy in Table 3.4 (with **same** = **same-operation-as**, **inverse** = **inverse-operation-of** and **super** = **super-operation-of**).

However, for optimization it is not sufficient to categorize the schema evolution operations based on just their functionality. It is important to also know the parameters, i.e., the context in which these operations are applied. Table 6.4 presents the schema element relationships.

While the *operation properties* (Table 6.2) and the *context properties* (Ta-

Schema Relation	Description
definedIn	gives the scope for all schema elements (from ODMG).
extendedBy	gives the inheritance relationship of schema elements of the type `Class` (from ODMG).
same-as	gives the identity of a class or a property based on unique name in given scope (CHOP extension).
aliasedTo	gives the derivation of a schema element from another element through a series of name modifications (CHOP extension).

Table 6.4: Classification of Schema Element Relations.

ble 6.4) provide *necessary* criteria for when an optimization function can be applied, they are not always sufficient in the context of a sequence. Here we briefly summarize the relationships of operations in a sequence.

- **Schema-Invariant Order Property.** When operation `op1` is **sameAs** `op2`, we identify the **schema-invariant-order** property as:
  - For two **capacity-augmenting** and **capacity-reducing** operations, `op1` is in **schema-invariant-order** with `op2` if the order of their parameters is the **same**.
  - For **capacity-preserving** operations, `op1` is in **schema-invariant-order** with `op2` if the order of their parameters is **reversed**.
- **Object-Invariant-Order Property** - `op1` is **object-invariant-order** with `op2` if `op1` is **capacity-augmenting** and `op2` is **capacity-reducing** and in the sequence of evolution operations the **capacity-augmenting** operation appears prior to the **capacity-reducing** operation. There is no specific **object-invariant-order** for the **capacity-preserving** operations.

- **Dependency Property** - The schema elements used as parameters by the two operations `op1` and `op2` being considered for optimization must not be referred to by any other operation which is placed between the two operations in the sequence.

6.1.2 The CHOP Optimization Functions

The integral component of the CHOP optimization algorithm are the optimization functions that can be applied to pairs of schema evolution operations within the context of their resident schema evolution operation sequence. In this section we present the general description of an optimization function and three instantiations of the optimization functions (*merge*, *cancel* and *eliminate*), that we have formulated for the optimization of the primitive set of schema evolution operations given in Section 3.2.2.

An Optimization Function

The crux of the CHOP optimization algorithm is an optimization function which takes as input two schema evolution operations and produces as output zero or one schema evolution operation, thereby reducing the sequence of the schema evolution operations. Formally, we define an optimization function as follows:

Definition 1 *Given a schema evolution operation sequence* Σ, $\langle$`op1, op2 ... opn`$\rangle$ *with* op_i *before* op_j $1 \leq i \leq j \leq n$, *an optimization function* F_Σ *produces as output an operation* `op` *which replaces* op_i. *The operation at index position* `j` *is set to a* `no-op`. *The operations* `op`, op_i *and* op_j *are either schema evolution*

primitives as described in Table 6.1 or complex evolution operations as defined in Section 6.1.2.

A major requirement for the CHOP optimization is to reduce the number of schema evolution operations in a sequence such that the final schema produced by this optimized sequence is consistent and is the same as the one that would have been produced by the unoptimized sequence for the same input schema. Thus, an optimization function must not in any way change the nature of the schema evolution operations or the order in which they are executed. Towards that goal, any optimization function must observe several properties we now list.

Invariant-Preserving-Output Operations. Schema evolution operations guarantee the consistency of the schema and the database by preserving the invariants defined for the underlying object model [BKKK87]. An important property of the optimization function therefore is for its output (any of its output operations) to also preserve the schema consistency by preserving the invariants defined for the object model.

Schema-State Equivalent. Above all an optimization function must guarantee correctness, i.e., the schema produced by output of the optimization function (optimized sequence) must be the same as the schema produced by the unoptimized sequence when applied to the same input schema. This property can in fact be proven formally (refer [CNR99]).

Relative-Order Preserving. As discussed in Section 6.1.1, the order in which the schema evolution operations appear with respect to one another in a sequence is relevant to the application of an optimization function. This **relative order** of an operation $\texttt{op}_i$ in a sequence is defined by its index, $\texttt{i}$, with respect to the index of the other operations. For example, if $\texttt{i} < \texttt{j}$, then $\texttt{op}_i$ is ***before*** $\texttt{op}_j$ in the sequence, denoted by $\texttt{op}_i < \texttt{op}_j$. Operations executed out-of-order can cause unexpected variance in the final output schema. For example, consider two operations in a sequence with the order as given here: $< \texttt{DA(C,a),AA(C,a)} >$. When executed in the order given the attribute a is first deleted from the class C and then re-added. However, all of the information stored in the attribute a is lost. Now, switching the order of execution of the two operations leads to a very different schema.

Thus, it is essential for an optimized sequence to preserve the **relative-order** of the input sequence.

Using the above stated properties, we now refine the definition of an optimization function as follows:

Definition 2 (Optimization Function.) *Any optimization function in CHOP defined as in Definition 1 must be* **invariant-preserving**, **schema-state-equivalence preserving** *and* **relative-order preserving**.

For the CHOP approach, we define three such optimization functions, Merge, Eliminate, and Cancel.

The Merge Optimization Function

The time taken for performing a schema evolution operation is largely determined by the page fetch and page flush times [FSS+97]. In our proposed CHOP approach we amortize the page fetch and flush costs over several operations by collecting all transformations on the same set of objects and performing them simultaneously [2].

A **complex operation** denoted by $< \texttt{op}_1, \; .... \; \texttt{op}_k >$ with $\texttt{k} \geq 2$ is defined as a collection of schema evolution operations for the same class which affect the same set of objects, i.e., it is possible to perform all the object transformations for these operations during the same page fetch and flush cycle. For two complex operations, $\texttt{op1} = < \texttt{op}_i .... \; \texttt{op}_j >$ and $\texttt{op2} = < \texttt{op}_m .... \; \texttt{op}_n >$, the operation pairs $(\texttt{op}_j, \texttt{op}_m)$ and $(\texttt{op}_n, \texttt{op}_i)$ are termed **complex-representative pairs**.

Definition 3 **Merge** *is an optimization function (Definition 2) that takes as input a pair of schema evolution operations, either primitive or complex,* `op1` *and* `op2`*, and produces as output a complex operation* $\texttt{op3} = < \texttt{op1}, \; \texttt{op2} >$*. If one or both of the input operations are a complex operation, e.g.,* $\texttt{op1} = <\texttt{op}_i, \; ... \; \texttt{op}_j >$ *and* $\texttt{op2} = <\texttt{op}_m, \; ... \; \texttt{op}_n >$*, then a relative order within the complex operations* $\texttt{op}_3$ *is maintained such that the output operation* $\texttt{op3} = <\texttt{op}_i, \; ..., \; \texttt{op}_j, \; \texttt{op}_m, \; ..., \; \texttt{op}_n >$*. The input operations* `op1` *and* `op2` *must satisfy:*

- **Context Property**

[2] This merge of operations relies on the underlying OODB system to be able to separate the schema evolution operation into a schema change at the Schema Repository level and into object transformations at the database level.

- *If* `op1` *and* `op2` *are related by the* **same-operation-as** *property, then their context parameters must be* **definedIn** *same scope.*
- *If* `op1` *and* `op2` *are related by the* **super-operation-of** *property, then for the sub-operation the* **definedIn** *scope of the context must be the* **sameAs** *the context of the super-operation.*
- *If* `op1` *and* `op2` *are related by the* **inverse-operation-of** *property, then the context of* `op1` *must be* **sameAs** *the context of* `op2` *and* **definedIn** *same scope.*

- **Dependency Property** *must hold.*

When one or both of the input operations `op1` *and* `op2` *are complex, then all the merge conditions given above must be satisfied by at least one pair of operations in the complex operation. This is the* **complex-representative pair** *and enforces the an ordering of the complex operation.*

For example, given an operation that adds a `name` attribute to a class called `Employee` and a subsequent operation that adds a `age` attribute to the same class `Employee`, we can merge the two operations they are related by the **same-operation-as** property and their context parameters are **definedIn** the same scope, i.e., `Employee` for both operations is in the same schema and the attributes `name` and `age` are being added to the same class `Employee`. Lastly, the **dependency** property holds as there are no operations between the index positions of the two operations.

A complex operation is thus a sub-sequence of schema evolution operations and other optimization functions (cancel and eliminate) can be ap-

plied on the primitive schema evolution operations inside of a complex operation. However, the merge optimization function itself cannot be applied inside a complex operation to prevent infinite recursion.

The Eliminate Optimization Function

In some cases a further optimization beyond merge may be possible. For example, while it is possible to merge `DA(Employee, name)` and `DC(Employee)` the execution of `DC(Employee)` makes the prior execution of `DA(Employee, name)` redundant. Hence, some operations may be optimized beyond a merge by being completely **eliminated** by other operations, thus reducing the transformation cost by one operation.

Definition 4 Eliminate *is an optimization function as defined in Definition 2 that takes as input a pair of schema evolution primitives* `op1` *and* `op2` *and produces as output* `op3`*, such that* `op3 = op1` *if* `op1` = **super-operation-of** *(*`op2`*) or* `op3 = op2` *if* `op2` = **super-operation-of** *(*`op1`*). The input operations* `op1` *and* `op2` *must satisfy:*

- **Operation Property** *such that either* `op1` = **super-operation-of** *(*`op2`*) or* `op2` = **super-operation-of** *(*`op1`*),*
- **Context Property** *such that the* **definedIn** *scope of the sub-operation is* **sameAs** *the context parameter of the super-operation, and*
- **Dependency Property** *must hold.*

The Cancel Optimization Function

In some scenarios further optimization beyond a merge and eliminate may be possible. Some schema evolution operations are inverses of each other, for example, `AA(Employee, age)` adds an attribute and `DA(Employee, age)` removes that attribute. A **cancel** optimization thus takes as input two schema evolution operations and produces as output a no-op operation, i.e., an empty operation that does nothing.

Definition 5 Cancel *is an optimization function as in Definition 2 which takes as input a pair of schema evolution primitives* `op1` *and* `op2` *and produces as output* `op3`*, where* `op3` = `no-op`*, an empty operation, assuming the input operations* `op1` *and* `op2` *satisfy:*

- **Operation Property** *such that* `op1` *and* `op2` *are related by the* **inverse-operation-of** *property,*
- **Context Property** *such that* `op1` *and* `op2` *are* **definedIn** *the same scope and* `op1` *is* **sameAs** `op2`*,*
- **Schema-Invariant-Order Property** *for capacity-reducing operations must hold,*
- **Object-Invariant-Order Property** *must hold, and*
- **Dependency Property** *must hold.*

6.1.3 CHOP Optimization Strategy

The CHOP optimization algorithm iteratively applies the three classes of optimization functions **merge**, **eliminate** and **cancel** introduced in Section 6.1.2 until the algorithm terminates and a minimal solution is found.

However, before we can address the issue of minimality it is necessary to examine two issues: (1) if one or more functions are applicable, is choosing the right function essential? and (2) when there are more than one pair of operations that can be optimized, is choosing the right pair essential?

Choosing the Right Optimization Function. We note that the conditions under which the **merge** optimization function can be applied is a *superset* of the conditions under which an **eliminate** or a **cancel** can be applied, while the conditions under which a **cancel** and an **eliminate** can be applied are mutually exclusive. Thus, often a **merge** can be applied to a pair of operations where either an **eliminate** or a **cancel** can be also applied. However, as these optimizations offer different degrees of reduction for a pair of schema evolution operations (with **merge** offering the least and **cancel** the most), choosing the optimization function that offers the most reduction is desirable.

We can however formally show that doing a **merge** where a **cancel** or an **eliminate** is also applicable does not prevent the application of a **cancel** or an **eliminate** during the next iterative application of these functions. A formalization of this property and its proof is given in [CNR00].

Operation Dependencies and Optimization Functions. An important criteria for the successful application of any of the three optimization functions is that the **Dependency Property** as given in Section 6.1.1 must hold. That is, there must be no reference to the schema elements used as parameters in the two operations `op1` and `op2` being considered for optimization by any other operation which is placed between the two operations in the sequence. However, the order in which the pairs of operations are selected can have an effect on this dependency.

Consider a sequence of three operations `op1`, `op2` and `op3`. Consider that the pairs (`op1`, `op2`) and (`op2`, `op3`) can be immediately optimized while a successful optimization of the pair (`op1`,`op3`) requires removing the dependency operation `op2`. In this case, there are two possibilities for applying the optimization functions on the pairs of operations. We could either apply the respective optimization functions on the pair (`op1`, `op2`) and then on the pair (`op2`, `op3`) [3] and not be concerned about the optimization possibility between `op1`and `op3`. Or we could first apply the optimization function on the pair (`op2`, `op3`), reduce the dependency `op2` and then optimize the pair (`op1`, `op3`). However, as before our goal is to achieve the *maximum* optimization possible.

We can formally show that the order of selection for pairs of schema evolution operations in a sequence for the application of one of the three optimization functions does not prevent the achievement of *maximum* optimization [CNR99].

[3] Note that in some cases `op2`may not exist any more and hence optimizing (`op2`, `op3`) may no longer be possible.

Confluence. While the main goal for the optimization is to achieve maximum optimization possible in an effort to reduce schema evolution costs, we also want to keep the overhead of optimizing to a minimal. However, there are multiple permutations and combinations of the optimization functions and the pairs of schema evolution operations that can potentially achieve the maximum optimization. Enumerating all the possible choices prior to selecting one for execution results in an exponential search space. This overhead from enumerating these choices alone would cancel any potential savings achieved by the optimization.

However, based on the function properties in [CNR99], we can show that all possible combinations of optimization functions for a given sequence converge to one **unique minimal**, thereby eliminating the need to enumerate all the possible choices. The following states the theorem of confluence. The proof for this is given in [CNR99].

Theorem 8 [Confluence Theorem]: *Given an input schema evolution sequence, Σ_{in}, all applicable combinations of optimization functions f_i produce minimal resultant sequences Σ_i that are identical.*

6.1.4 Experimental Validation

We have conducted several experiments to evaluate the potential performance gains of the CHOP optimizer. Our experimental system, CHOP, was implemented as a pre-processing layer over the Persistent Storage Engine (PSE Pro2.0). All experiments were conducted on a Pentium II, 400MHz, 128Mb RAM running WindowsNT and Linux. We used a `payroll` schema

[CNR99]. The schema was populated with 5000 objects per class in general or are otherwise indicated for each individual experiment. Due to lack of availability of a benchmark of typical sequences of schema evolution operations, the input sequences themselves were randomly generated sequences.

The applicability of CHOP is influenced by two criteria, the performance of the optimized vs the unoptimized sequence of schema evolution operations, and the degree of optimization achievable on average by the optimization functions of CHOP. Here we present a brief summary of our experimental observations. The details can be found in [CNR99].

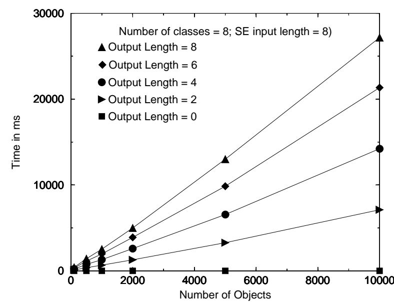


Figure 6.1: Best and Worst Case Sequence Times w/o Algorithm Overhead for Input Sequences of Length 8 on the Sample Schema.

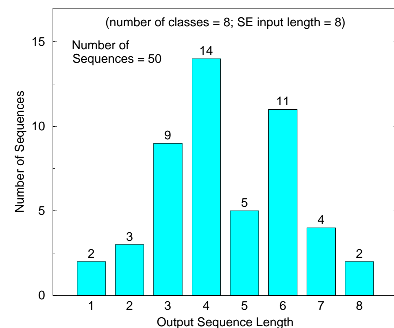


Figure 6.2: Distribution: Number of Classes = Sequence Length.

- The SE processing time for a sequence is directly proportional to the number of objects in the schema. Hence, for larger databases we can potentially have larger savings.

- The optimizer algorithm overhead is negligible when compared to the overall cost of performing the schema evolution operations themselves. Thus our optimization as a pre-processor offers a win-win

solution for any system handling sequences of schema changes (Figure 6.1).

- The degree of optimization increases with the increase in the number of class-related operations in the sequence. Hence, depending on the type of sequence, major improvements are possible (Figure 6.2).

- A random application of the optimization functions on the same sequence resulted in the same final sequence of schema evolution operations.

- We have experimentally tested that on a small-sized database of 20,000 objects per class, even the removal of a single schema evolution operation on a class already results in a time saving of at least 7000 ms. This time savings is directly proportional to the number of attributes and the extent size of a class thus offering huge savings for today's larger and larger database applications.

6.2 Summary

In this chapter we have presented the first optimization strategy for schema evolution sequences. CHOP minimizes a given schema evolution sequence through the iterative elimination and cancellation of schema evolution primitives on the one hand and the merging of the database modifications of primitives on the other hand. Important results of this work are the proof of correctness of the CHOP optimization, a proof for the termination of the iterative application of these functions, and their convergence to a unique

and minimal sequence. We have performed experiments on a prototype system that clearly demonstrate the performance gains achievable by this optimization strategy. For random sequences an average optimization of about 68% was achieved.

CHOP in its current form does not take into account the presence of OQL queries that are an inherent part of the a SERF template. This is a potential future work project that would extend the basic techniques that we have presented here.

Chapter 7

Design and Implementation of the OQL-SERF System

In this section we present a brief overview of our implementation of the SERF Framework - OQL-SERF. A more detailed description can be found in [CJR98b]. OQL-SERF, a Java-based system is built using Object Design Inc.'s Persistent Storage Engine Pro 2.0 (PSE Pro 2.0) as the underlying OODB system [O'B97]. It uses the JDK 1.2 and has been implemented and tested on Windows NT and Linux (S.U.S.E 6.3). It is based on the ODMG standard. In particular, we used an extension of Java's ODMG binding and our own Java binding of the ODMG Schema Repository. A prototype of this system was demonstrated at SIGMOD 1999 [RCL+99].

7.1 System Architecture

Figure 7.1 presents the system design for OQL-SERF. PSE, a lightweight OODB written in Java [O'B97], has been chosen for SERF due to its portability to different platforms, the availability of a free version and its conformance to the Java ODMG binding.

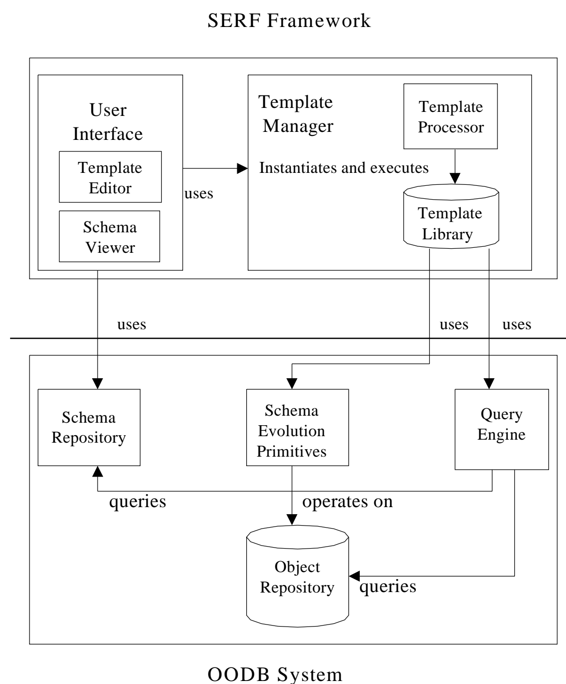


Figure 7.1: Architecture of the SERF Framework

In a persistent storage system, such as PSE, it is assumed that the schema representation, data, applications and the links between them are all held as objects in persistent storage. While PSE offers most OODB features, it does not explicitly define a Schema Repository as per the ODMG standard. It also does not have the requisite schema evolution support and its query

interface does not meet the requirements of a query language for SERF.

Thus as part of the OQL-SERF implementation, we have addressed these limitations and built:

- A fully operational ODMG compliant Schema Repository.
- A complete schema evolution facility that preserves the ODMG object model and does changes at a finer granularity, thus allowing for in-place evolution.
- An OQL Query Engine for querying the objects in PSE.

In the future we would expect an underlying OODB realizing SERF to provide these basic capabilities. In this section we hence do not go into the details of the above extensions (see [CJR98b]). Rather, we focus on describing the framework modules that are the core of SERF.

7.2 SERF Framework Modules

The SERF Framework Modules are the core components that need to be provided by any system realizing the SERF Framework. Sections 7.2.1 and 7.2.3 describe the functionality and support needed for SERF templates and transformations.

7.2.1 Template Module

The Template Module provides all of the functionality for storing, retrieving and executing templates. Figure 7.2 shows the architecture for the Template Module in OQL-SERF.

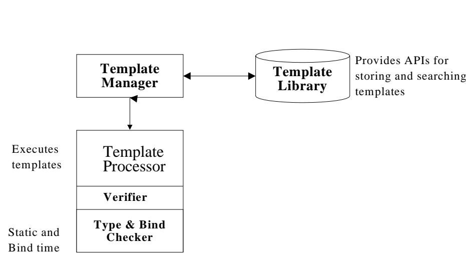


Figure 7.2: The Template Module.

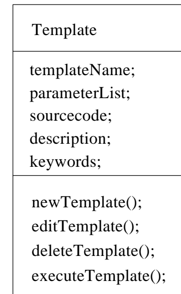


Figure 7.3: The Template Class.

The Template Manager. The Template Manager is the public interface of the template module. Through the template manager the user can retrieve, edit and execute already existing templates as well as create and store new templates.

The Template Processor. The Template Processor executes a template. Figure 7.4 shows the steps, beginning with the user supplying the input parameters. A *type-check* ensures that the types of the parameters match and they exist in the system as well as the correct number of parameters are supplied by the user. This is followed by a *bind-check* that checks the existence of these actual parameters in the schema on which they are being applied by accessing the Schema Repository. If the checks are successful, the SERF template is instantiated using these parameters by replacing each variable with its bound parameter. The instantiated template now corresponds to pure OQL statements, i.e., we now call it an OQL transformation. The OQL Query Engine provides an interface for the syntax-checking, the *parsing* and the *execution* of the OQL transformation.

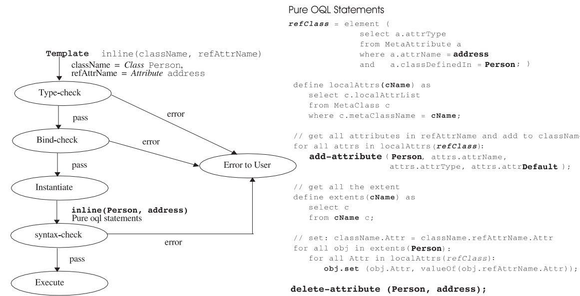


Figure 7.4: Steps for the Execution of a Template

7.2.2 Template Library

The SERF templates for a particular domain are collected in a Template Library. A Template Manager at any given time can manage multiple Template Libraries. For this reason `libraryName.templateName` gives the complete path for storing and retrieving a template. Within a template library each template is represented by a corresponding template object [1], an instance of the TemplateClass (see Figure 7.3). Each template object hence contains a ***name***, ***description***, set of ***input parameters***, list of ***keywords*** and ***oql source*** for its template.

The Template Library currently provides the users (via the Template Manager) the capability to search for a template in the library via simple keyword search on all stored parameters of the template such as the input and output parameters, the name, and the description.

[1]We distinguish between a template and a SERF template. Here a template object implies an instance of the TemplateClass and a SERF template is the source code that is part of this instance.

7.2.3 User Interface

For OQL-SERF version 1.0, we have designed a graphical user interface (GUI) as a front-end to the Template Manager. It provides a Template Editor with syntax highlighting that allows the user to create a new or edit an existing template as shown in Figure 7.5. The Save option walks the user through the steps of providing the *name*, *list of formal parameters*, *description*, and *keywords* for the SERF template object before it can be stored and managed by the template library. It also provides a graphical interface for viewing, searching and retrieving the templates stored in the Template Library. The GUI allows the user to utilize a schema viewer to select the parameters for instantiating a template (see Figure 7.6). The user can then view the schema graph before and after a transformation has been applied to the schema thus giving the user a chance to verify if this was indeed the desired transformation.

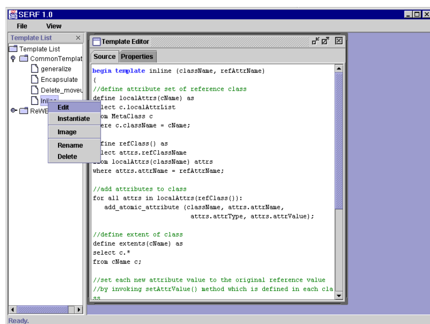


Figure 7.5: The Template Editor.

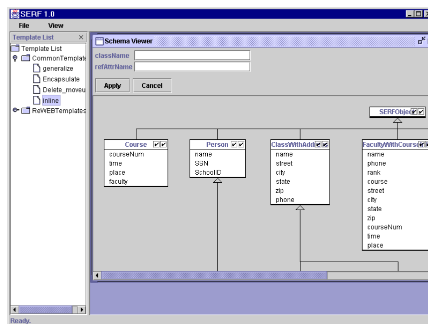


Figure 7.6: The Schema Viewer.

7.3 Summary

In this chapter we have presented the architecture and the implementation details of our prototype system, OQL-SERF. This system was demonstrated at SIGMOD99 [RCL+99]. The key advantage of this architecture is that it provides enhanced schema evolution capabilities without requiring the underlying system to change.

Chapter 8

Case Study of Different Schema Transformations

We have presented SERF as a system that gives users:

- Extensibility: Allows users to define new transformations so they are not limited by one fixed taxonomy of schema evolution primitives provided by the underlying OODB;
- Flexibility: Allows users to define semantics of their choice for a transformation and does not limit them to the pre-defined semantics set forth by the underlying OODB.
- Re-usability: Allows the users to move away from writing ad-hoc programs for complex transformations and instead offers the users re-usability of their OQL code via the templates as well as by building templates composed of other templates.

Several researchers have looked at simple and complex schema transformations in the context of schema evolution [Ler00, Bré96], schema integration [BLN86, MIR93], and re-structuring [BB99, DKE94, DK97] in general. A large subset of these transformations involves changing more than one class, such as *inline* which inlines two classes dependent on the reference attribute, or *merge* which combines two classes based on a variety of criteria. We have been successful at modeling the transformations proposed in prior works using SERF templates. The goal of the case study presented here is to show that SERF transformations can not only handle the transformations presented in literature (*inline*, *encapsulate*, *split*, *merge-union*, *merge-difference*) but also capture different semantics specified by the user. In this section we present the *merge* transformation and show how a user can specify templates each with different semantics for creating the *merged* class. We also present examples of templates that are now applied for the re-structuring of web pages.

8.1 The Merge Transformation

Lerner [Ler00] defines *merge* as a compound type change involving type deletion. *Merging* deletes two or more object types and creates a new type that represents the integration of the deleted types [Ler00]. The semantics of the merge, such as the attributes based on which the objects can be combined, are built into the merge compound change. However, we note that there are several different possibilities for the structural definition of the new type as well as for its object creation. For example, the structure

of the new class can be defined by doing a union of the properties of the two source classes or by an intersection of the attributes of the two classes. Similarly, object creation can be specified by a Cartesian product, by a simple join of one single property or by possibly a more complex join. By previous approaches in the literature focusing on complex transformations [Bré96, Ler00], the merge has generally been defined as ***one fixed transformation***. Below in our case study we first show how using templates we can achieve the semantics as defined by them and then can apply new semantics at both the structural level and the object transformation level flexibly within SERF. In the merge transformation case study below we show three different semantics for achieving the structural definition of the new class (merge, difference, intersection) and then we will show two possibilities for doing the object transformations for the merge transformation.

8.1.1 The Merge-Union Template

Figure 8.1 represents the semantics of the merge-union process using an example. Here the structure of the new class `MergePapers` is defined by a union of the properties of the two source classes `Authors` and the `Papers`. Figure 8.3 shows the template that is used for creating the merge-union class `MergePapers`. This template takes five parameters: the two source classes; the name of the new class (merge-union) class; and the join operation. In our example, creation of the new objects for the `MergePapers` class is specified by the join of the `Authors` objects and the `Papers` objects such that `Authors.AuthorName = Papers.AuthorName`.

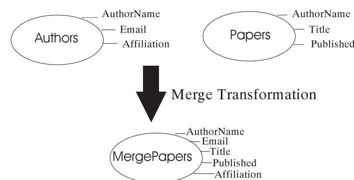


Figure 8.1: Merge-Union: The Structure of the New Class given by Union of the Properties of the Two Source Classes `Authors`, `Papers`.

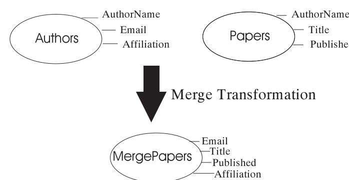


Figure 8.2: Merge-Difference: The Structure of the New Class given by Difference of the Properties of the Two Source Classes `Authors`, `Papers`.

8.1.2 The Merge-Difference Template

The Merge-Union template shows one possible semantic definition for the structure of the new class. Another possible semantic for a merge is to create a new class by taking a difference of the sets of properties of the two given classes. That is, if a property exists in both of the given classes, then it is not a property of the new class. Figure 8.2 pictorially represents the merge-difference semantics. Here the structure of the new class `MergePapers` has the properties `Email`, `Affiliation`, `Published` and `Title`, the difference set of the properties of the two source classes `Authors` and `Papers`. Figure 8.4 shows the template that is used for creating the merge-difference class `MergePapers`. The object transformation for the template is based on the same join pair as above.

It can be observed by comparing the templates that the only a few lines of OQL code have been modified as marked in Figure 8.4. Thus in SERF *extensibility* is often obtained with little effort.

```
begin template merge_union(Class className1, Class className2,
                           String newCName, Attribute attr1,
                           Attribute attr2, String join_op)
{
    // Retrieve all attributes of class1 and class2
    define uAttributes(cName1, cName2) as
        select distinct c.localAttrList
        from MetaClass c
        where c.className = cName1 or
            c.className = cName2;

    // Create the new merge class
    add_class ($newCName);

    // Add all the attributes to it
    for all attr in uAttributes($className1, $className2):
        add_attribute($newCName, attr.attrName,
                      attr.attrType, attr.attrDefault);
    // Get the extent of a class
    define extent(cName) as
        select c
        from cName c;

    // Do the object transformation
    define new_extent () as
        select obj1.*, obj2.*
        from extent($className1) obj1, extent($className2) obj2
        where
        obj1.$attr1 $join_op obj2.$attr2;

    for all obj in new_extent ():
        $newCName(obj);
}
end template
```

Figure 8.3: The `Merge-Union` Template.

8.1.3 A More Complex Merge Template

The difference between the `Merge-Union` and `Merge-Difference` templates is in the OQL statements that acquire the property set that is to be

```
begin template merge_diff(Class className1, Class className2,
                          String newCName, Attribute attr1,
                          Attribute attr2, String join_op)
{
    // Basic statement to get attributes of a class
    define attributes(cName) as
        select c.localAttrList
        from MetaClass c
        where c.className = cName;

    // Retrieve the difference attributes of class1
    define diffAttributes(cName1, cName2) as
        select attrs.*
        from attributes(cName1) attrs
        where not (attrs exists in attributes(cName2));

    // Create the new merge class
    add_class ($newCName);

    // Add all the attributes to it
    for all attr in diffAttributes($className1, $className2):
        add_attribute($newCName, attr.attrName,
                  attr.attrType, attr.attrDefault);
    // Get the extent of a class
    define extent(cName) as
        select c
        from cName c;

    // Do the object transformation
    define new_extent () as
        select obj1.*, obj2.*
        from extent($className1) obj1, extent($className2) obj2
        where
        obj1.$attr1 $join_op obj2.$attr2;

    for all obj in new_extent ():
        $newCName(obj);
}
end template
```

Figure 8.4: The `Merge-Difference` Template.

```
begin template merge_common(Class className1, Class className2,
                            String newClassName, Attribute attr1,
                            Attribute attr2, String join_op)
{
    // Get the extent of a class
    define extent(cName) as
        select c.*
        from c in cName;

    // Do the object Transformation
    define new_extent () as
        select obj1.*, obj2.*
        from extent($className1) obj1, extent($className2) obj2
        where obj1.$attr1 $join_op obj2.$attr2;

    for all obj in new_extent ():
        $newClassName(obj);
}
end template
```

Figure 8.5: The `Merge-Common` Template - Shows the Common Code between the `Merge-Union` and the `Merge-Difference` templates.

added to the new class, `MergePapers`. The remaining, common set of OQL statements can be abstracted into a subroutine represented by another template as shown in Figure 8.5.

8.2 ReWEB - Applying SERF for Web Transformations

As part of the SERF work, we have developed *Re-WEB* [CRCK98, CCR00][1], a SERF-based system that is now applied for re-structuring web sites. In this section, we present an example of web site re-structuring in ReWEB,

[1]This work was done in collaboration with Li Chen a Ph.D student at WPI.

while most of the details can be found in [CCR00]. All displayed web pages have been generated automatically by our Re-Web system.

First, assume the web administrator designs the web site structure of a university's computer science department. The index *Home* page provides three categories of information: Faculty members, Courses and ClassRooms, each of which has a link referring to its respective list. The *FacultyList* index page, under the *Faculty* sub-directory relative to the *Home* page, lists all the faculty members of this department. Each faculty has a link to a separate page named by the name of the faculty (for example, *Elke* could be the name for her homepage), the personal information, as well as a link to each course she teaches. Each *Course* page (that the department provides) has *Faculty* links to all the instructors who are offering the course. With this design of homepages within the department's web site, the web administrator basically captures this web semantics by designing the original OODB schema as depicted in Figure 8.6. The desired web pages can be generated accordingly (see the lower part of Figure 8.6).

In Figure 8.6, we assume there are objects that populate the database already. Assume there is a *Course* object with courseNum "CS561", two faculty members are offering this course, one of them is Elke A. Rundensteiner, whose personal information is included in a *Faculty* object. The course offered by her is held in Fuller Lab 320 every Thursday night from 5:30 to 8:20 pm. The classroom location information is encapsulated in a *ClassRoom* object. Thus there exists a bi-directional link between the *Faculty* object and the *Course* object, modeling a (zero to multiple) relationship. Also there is a one-way relationship from the *Course* object to the *ClassRoom*

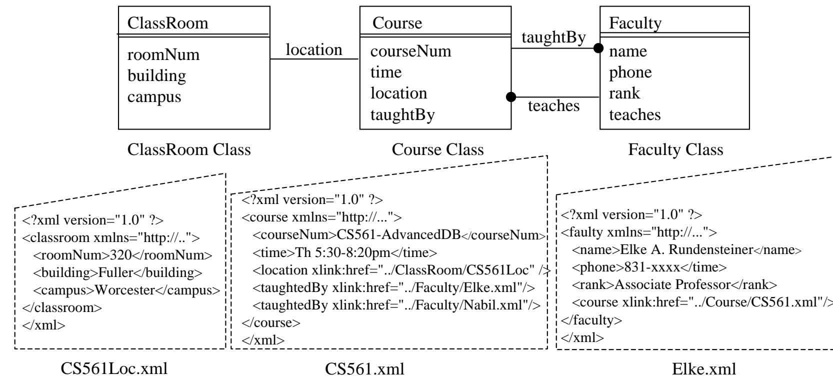


Figure 8.6: The ODMG Data Model and Corresponding XML Files for the Original Web Site.

object. Each of these three objects has an associated type in the OODB and the relationships among them follow the OODB schema design.

In the underlying database, all objects of each class are maintained in the class extent and all classes are registered by the schema dictionary. Thus by scanning the schema dictionary, a set of XML files are generated by the Re-Web system. Each of them corresponds to one object from the OODB. We show some of the generated XML files in Figure 8.6. They are *CS561.xml* under the new created subdirectory of *Course*, *Elke.xml* under the subdirectory of *Faculty* and *CS561Loc.xml* under the subdirectory of *ClassRoom*. The XLinks within the XML files capture the relationships between the objects and simulate the navigation mechanism.

Lastly, based on these XML files, the corresponding home pages are then generated using the LotusXSL processor [IBM98] provided by IBM (see Figure 8.7). On the top of each page, there is a navigation bar indicating the categories of home pages. Each category has an index page that

contains the list of pages that falls into this category. For example, by clicking on the *Faculty* bar, a *Faculty_List* index page is requested containing all links to the faculty members.

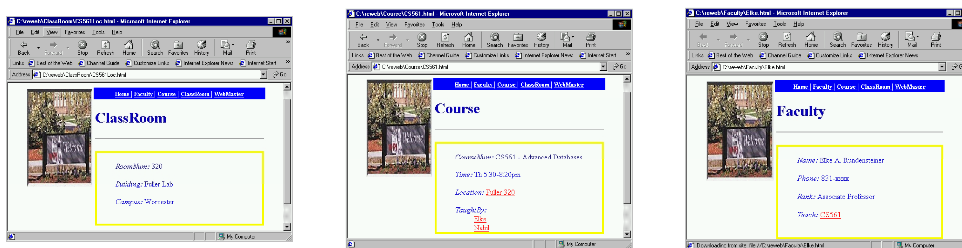


Figure 8.7: The Generated Example Home Pages of Original Web Site

However, this web site schema may not meet the user's requirements and the user may instead desire a web view of the same data but with information in a more compact format. For example, by checking a faculty's home page, the user would like to be able to know about not only the personal information of this faculty, but also all the courses she is teaching as well as the location of that course within the same page. Thus the web site needs to be re-structured. The supporting database schema should be as depicted in the left part of Figure 8.9. After performing the desired view transformation using the chain nesting transformation depicted in Figure 8.8 with the help of the SERF subsystem, the underlying database is re-structured to correctly reflect the desired structure of the web site.

Again, using the WebGen subsystem, the restructured database schema and its transformed objects are dumped out and represented in XML format (as shown at the right part of Figure 8.9). The screen-dump of a resulting example homepage is shown in Figure 8.10. The previous home page

```
begin template nested-convert-to-literal ( Class inliningClassName,
                                           Attribute flattenAttrName,
                                           Integer nestLevel)
{

    // inline the next level complex attribute into the specified inlineClass.
    inline ($inliningClassName, $flattenAttrName)

    // decide for a class which of its attibutes is complex and to be flattened.
    define findAttrToFlatten (className) as:
       select a
       from MetaAttribute a, MetaClass c
       where a.classDefinedIn = className
       and a.attrType = c.name
       and nestLevel != 0

    // starting at the inliningClass all complex attributes are flattened.
    for attr in findAttrToFlatten ($inliningClassName):
        nested-convert-to-literal($inliningClassName, attr, $nestLevel-1);
}
end template
```

Figure 8.8: Nested Inline Template by Re-using Basic Inline Template.

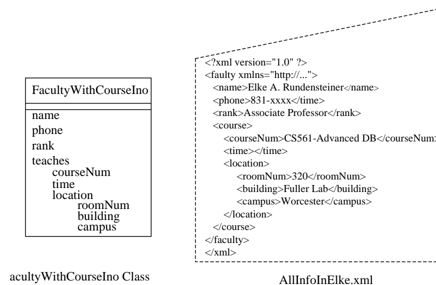


Figure 8.9: The ODMG Data Model and Corresponding XML Files for Desired Web Site

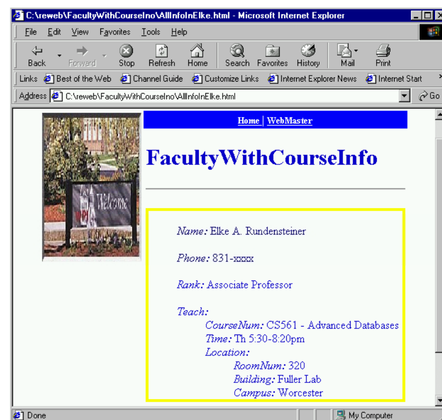


Figure 8.10: The Generated Example Home Pages of Desired Web Site

for the faculty named “Elke” has been transformed and now directly embeds all the information collected along the chain from itself to its *Course* links and from there to their respective *Classroom* links.

The power of Re-WEB lies in its capabilities to do complex transforma-

tions, i.e., those defined in the template library or user-defined transformations. The example presented above is one example that shows the ease with which web pages can be re-structured using Re-WEB. The above example however, requires that a link or a relationship between the two web pages (or classes) must exist for its successful execution. However, a user may wish to combine information from two disjoint web pages (no link) and present it in one web page.

Figure 8.3 can be used to combine two disjoint classes and produce a view that represents a union of the two classes. Figure 8.4 shows a template which can combine two disjoint classes and produces a view that represents a difference of the properties of the two classes. This view can now be generated by WebGen to produce a composite web page between two otherwise disjoint web pages.

8.3 Summary

In order to validate the basic concepts of SERF, we have presented in this chapter two case studies. The first highlights the ability to express different semantics for the same operation using the SERF templates. The second case study shows the utility of the SERF systems as a web site re-structuring tool.

Chapter 9

Related Work

9.1 Specifying Schema Change

Simple and Complex Schema Evolution. Schema evolution is a problem that is faced by long-lived data. The goal of schema evolution research is to allow schema evolution mechanisms to change not only the schema but also the underlying objects to have them conform to the modified schema. One key issue in schema evolution is understanding the different ways of changing a schema. The first taxonomy of primitive schema evolution operations was defined by Banerjee *et al.* [BKKK87]. They defined consistency and correctness of these primitives in the context of the Orion system. Most systems today such as Itasca [Inc93], GemStone [BMO+89], ObjectStore [Obj93], and O_2 [Tec94] provide similar evolution support for their underlying object model.

In recent years, the advent of more advanced applications has led to the need for support of complex schema evolution operations. [SHT+77,

FFHI72, Bré96, Ler00, Cla92] have investigated the issue of more complex operations. Much of the initial work [SHT$^+$77, FFHI72] focused on converting and re-structuring data from flat files to a database which often required complex conversions. Shu *et al.* [SHT$^+$77] proposed complex operations such as *SLICE* to flatten a hierarchical structure, *GRAFT* to merge two trees into one larger tree and *SELECT* to select trees based on some condition. More recently, [Ler00] has introduced compound type changes in a software environment, i.e., focusing only on the type changes and not the changes to the object instances associated with the modified type. The proposed compound type changes included *Inline*, *Encapsulate*, *Merge*, *Move*, *Duplicate*, *Reverse Link* and *Link Addition.*

Bréche [Bré96] proposed a similar list of complex evolution operations for O_2, i.e., which considered schema as well as object changes. Bréche shows that these advanced primitives can be formulated by composing the basic primitives that are provided by the O_2 system, and has shown the consistency of these advanced primitives. User-customized object migration functions in the (C-like) programming language of O_2 can however be provided. A SERF template is similar to these complex operations proposed by Breche [Bré96] in the O_2 framework. However, while they allow users flexibility for the data transformations (migration functions), the same flexibility is not available to the users at the schema level. Moreover, this set of primitives, now more complex, is not extensible, i.e., a user cannot add their own schema transformations. In summary, previous research in database schema evolution provides the users with a *fixed* set of schema evolution operations [FFM$^+$95, BKKK87]. Adding extensibility to schema

evolution is the focus of our effort.

Work similar to ours has been done in the context of *database transformation*. The schemas involved can be expressed in a variety of different database models and can be implemented using different DBMSs. Shu *et al.* [SHL75] introduced a new data translation language, CONVERT, for translating between source items and target items. Their work was however primarily focused on hierarchically structured data and their set of operations is also fixed. More recently, Davidson *et al.* define a new language (WOL) [DK97] for specifying such database transformations. In our work, we do not propose a new language. To the contrary, we combine a standard database query language and schema evolution services into one framework and strive to provide a consistent yet extensible environment for doing both the schema and the data transformations.

A key requirement of SERF is access to the system dictionary via OQL. For this, we expect the system dictionary (schema repository) of an OODB to be fully ODMG-compliant. If a given OODB is not ODMG compliant, then all SERF templates require modifications to utilize the specific notations of the proprietary schema repository of the OODB. This modification of SERF templates can be eliminated if the query language used in SERF is modified such that it allows uniform access of data and meta-data. One such query language *SchemaSQL* has been proposed by Lakshmanan *et al.* [LSS96]. *SchemaSQL* is an extension of SQL that allows for manipulation of data and meta-data in relational multi-database systems. This has the advantage of offering inter-operability for meta-data management in relational systems and is not bound to the low level implementation of the sys-

tem dictionary. We are currently looking at extending OQL to allow similar uniform data and meta-data access and thus provide transparency of the low-level implementation [SCR00]. XSQL [KKS92] is a another example of a query language that provides some of the features of *SchemaSQL* in an object-oriented context.

Evolution of Persistent Languages and Object Databases. Persistent languages have made transparent the lines between a programming language and a data store. However, persistent objects must also face their evolution. In [RW98], Ridegeway *et al.* explore the evolution of classes in JSPIN. Here they store class binaries in the database and evolution looks at binary compatibility and at evolving it. However, their implementation currently only supports replacement of one binary with another. PJama, another persistent language, also supports evolution [Dmi98]. They present an approach that modifies the byte-code and verifies its authenticity before making it persistent. One of their contributions is the incremental generation of the byte-code depending on the schema evolution method.

Miscellaneous. Peters and Ozsu [PO95] have introduced an axiomatic model that can be used to formalize and compare schema evolution modules of OODBs. They develop a formal basis for schema evolution research, and we use their notation and model to represent the concepts presented in this paper.

9.2 Optimization of Schema Evolution

Research in schema evolution has also studied the issue of when and how to modify the database objects to address such concerns as efficiency, availability, and impact on existing code. Research on this issue has focused on providing mechanisms to make data and the system itself more available during the schema evolution process, in particular deferred and immediate propagation strategies [FMZ94b, FMZ94a]. Such optimization strategies are orthogonal to the issue of expressiveness and extensibility of the schema evolution support and in principle either of these propagation strategies could be implemented for our framework. In [FMZ94b, FMZ94a], a deferred execution strategy is proposed for the O_2 database system that maintains a history of schema evolution operations for a class and migrates objects only when actually accessed by the user. This allows not only for high database availability but also amortizes the cost of the object transformations with that of a query lookup. However, no optimizations are applied to this sequence of schema evolution operation(s) and the performance of this deferred mechanism deteriorates as the set of queried objects grows larger. Our approach [CNR00], while primarily optimizing the immediate update mode, also complements the deferred mechanism by offering time savings as the queried set of objects and the number of schema evolution operations to be applied on it grows larger.

9.3 Behavioral Consistency

Schema evolution can cause both structural as well as behavioral (code) inconsistencies. Zicari *et al.* [DZ91], Navathe *et al.* [MNJ94] and others have explored the effects of schema evolution on the methods defined for a class, while others like Bergstein *et al.* [BH93] have looked at it from a software perspective in terms of doing code transformations when schema evolution occurs. Ozsu *et al.* [OPS+95] have developed TIGUKAT, an Objectbase for modeling the uniform semantics of behaviors on objects and have explored behavioral consistency for TIGUKAT. All of these above mentioned works focus on the correctness of an individual schema evolution operation, and thus are a pre-requisite for our work. None of them addresses, however, the problem of optimizing the sequence of one or even several schema evolution operations as done by our work.

9.4 Consistency Management

Consistency management is often done at runtime and is normally handled by transaction roll-backs. Work has been done towards providing behavioral consistency, i.e., the consistency of class methods under evolving environments [DZ91, MNJ94]. While there are similarities in that their algorithms are also detecting broken references, such approaches are also hard-coded. In SERF, we push these constraints/invariants to a higher level thus making them easy to update and also allowing for pre-execution verification checks.

Other approaches for consistency management in ODBs utilize programming language support [VD91, AH90] such as assertions and exception handling mechanisms in languages like C++, Java and Ada. Relational database systems (RDBMS) offer some additional support in the form of *triggers* but provide only roll-back semantics, i.e., if a constraint is not satisfied at the end of a transaction, then the entire transaction is rolled back [EN96]. Active database systems [BCVG86, LLPS91, BK90] provide event-condition-action (ECA) rules as a mechanism for detecting the occurrence of some event and responding to it by some action.

Research has also been done in consistency management for software process languages. Tarr *et al.* [TC98] have developed a consistency management system which allows for the specification of consistency conditions and the degree of inconsistency tolerable by the user. Much of the work in literature concentrates on detecting consistency violations and on the specification of consistency constraints. We now try to utilize theorem provers as a preventive measure to ensure that consistency violations do not occur.

9.5 Extensible Systems

Hürsch *et al.* [HS96] have proposed a framework that captures the dependency between different components in a schema and code. When a change occurs this dependency framework is used to formulate propagation patterns to maintain behavioral consistency. They use propagation patterns as a mechanism for maintaining programs. We propose here a similar ap-

proach. However, we utilize a declarative approach for specifying the constraints embedded within schema evolution primitive code. We utilize the notion of Contracts [Mey92] as first proposed by B. Meyer to specify the constraints in a declarative fashion. Formal verification [GSW95, ORS92, Bla98, GM93] techniques or more informal verification algorithms can be utilized to verify the contracts.

Part III

Sangam - Managing Change in Heterogeneous Databases

Chapter 10

Overview

Specifying and executing change on a source database is only one part of the equation for change management. Typically, a database will have many views, i.e., information that is derived from the source information. A change in the source may result in changes or perhaps even in the invalidation of the view. Views and maintenance of views with respect to source changes has been a lively topic in database research [ZGMHW95, GB95]. Much of the work in this area has been based on the assumption that both the source and the view are defined within the confines of the same data model, and often within the same database. Today this world is fast changing. Today many application engineers struggle to not only publish their relational, object or ASCII file data on the Web but to also integrate information from diverse sources. That is views are no longer in the same data model as the source. Systems must therefore now address the issue of propagation of change, where in, the derived information is not in the same data model as the source.

Current research addresses this problem by developing *apriori* algorithms [TIHW01] that rely on the underlying translation algorithm between the two data models. These *apriori* algorithms are tightly coupled to the *apriori* algorithms used to derive the view from the source. We find that because of this tight coupling, the *apriori* maintenance algorithms fall short whenever there is a change in the translation technique between the same two data models or if the maintenance algorithm is to be applied views and sources in different data models. To meet the needs of maintaining such across data model views, we consider current technology inadequate.

```
ITEM(id VARCHAR(10) NOTNULL, featured VARCHAR(100),
        location VARCHAR(200), mailBoxID VARCHAR(10),
        lastName VARCHAR(200), firstName VARCHAR(200))

MAIL(from VARCHAR(50), to VARCHAR(50),
        date VARCHAR(8), mailBoxID INTEGER
        CONSTRAINT fk_id FOREIGN KEY (mailBoxID)
        REFERENCES ITEM(mailBoxID))
```

Figure 10.1: An Example Relational Schema.

To illustrate this problem, consider the fragment of the XMark benchmark DTD [SWK+01] shown in Figure 10.2 that is mapped to a relational schema and data (Figure 10.1) via the basic inlining method [STZ+99]. The basic inlining method inlines as many descendants of an element as possible into a single relation [STZ+99]. Consider now that the DTD (Figure 10.2) is modified and a new subelement **reserve** is added to the `item` element. Figure 10.3 depicts the modified DTD. To ensure consistency of the relational storage, this change should be propagated to the relational

schema and data. As per the basic inlining method, this change would result in a new table `RESERVE` and a new column `item.reserve` in the table `ITEM`. The data needs to be updated accordingly. However, if instead the same XMark DTD were mapped to a relational schema via the shared inlining method [STZ+99], the **same change** in the DTD would now be **propagated differently** to the relational schema, resulting in only the addition of a new column `item.reserve` in the table `ITEM`. No new table `RESERVE` would be created under this scheme[1]

```
<!ELEMENT item (location,
          mailbox, name)>
<!ATTLIST item id ID #REQUIRED
          featured CDATA #IMPLIED>
<!ELEMENT location (#PCDATA)>
<!ELEMENT mailbox (mail*)>
<!ATTLIST mailbox id CDATA>
<!ELEMENT mail (from, to, date)>
<!ATTLIST mail text CDATA>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT name (firstName,
          lastName)>
<!ELEMENT firstName (#PCDATA)>
<!ELEMENT lastName (#PCDATA)>
```

Figure 10.2: A Fragment of the XMark Benchmark Schema [SWK+01].

```
<!ELEMENT item (location, mailbox,
          reserve)>
<!ATTLIST item id ID #REQUIRED
          featured CDATA #IMPLIED>
<!ELEMENT location (#PCDATA)>
<!ELEMENT mailbox (mail*)>
<!ATTLIST mailbox id CDATA>
<!ELEMENT mail (from, to, date)>
<!ATTLIST mail text CDATA>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT reserve (#PCDATA)>
```

Figure 10.3: Modified XMark Benchmark Schema [SWK+01]. A New SubElement `reserve` is Added. The New Additions are Shown in Bold.

To propagate a change we must now deal with the fact that (1) the view is not in the same data model as the source; and (2) there is no query tree that derives the view from the source. Rather there are special purpose

[1] We refer the user to [STZ+99] for more details on the basic and shared inlining methods.

transformations which are most often in the form of a hard-coded module, requiring different propagation algorithm for each change primitive, each transformation, each pair of source and view (target) data models.

We address this problem of across data model maintenance using a two-step approach. We first define a middle-layer integration framework and as part of that a generic mapping language - the cross algebra, that can define views such that there is no restriction that forces the view and the source data models to be the same. Based on this algebra, we define update propagation algorithms that enable the propagation of schema and data changes from the source data model to the target data model.

Roadmap. Chapter 11 describes the graph model (Sangam graph) that we use as our basic data model. Chapter 12 describes the cross algebra and the different techniques of composing them into cross algebra graphs. Chapter 13 describes the working implementation of the Sangam system and presents the experimental validation of the same. In Chapter 14 we present the set of change operations that can be applied on a Sangam graph, and show how local changes in the relational or XML model can be mapped into these. Chapter 15 describes our incremental update propagation algorithm that can propagate these graph changes through the cross algebra graphs. Chapter 16 presents our experimental results and Chapter 17 gives an overview of the related work.

Chapter 11

Data Model

A key component of any middle-layer framework is the data model. The choice of this common data model is crucial as the effectiveness of the middle layer is tightly coupled to the expressive power of the middle-layer data model. A common data model must therefore be (1) be expressive enough to structurally represent schemas from a variety of different data models such as the relational, XML or object models; and (2) be able to express a common subset of constraints, such as the order constraints in XML, participation constraints (relational and XML), and other referential constraints such as key and foreign key constraints. While we looked at existing off-the-shelf data models we found that they did not adequately support the requirements for a common data model. For example, while the XML model satisfies the expressiveness property, it does not provide adequate support for key and foreign key constraints [1]. Similarly, the rela-

[1]The XML Schema specification given in May 2001 does provide support for keys and keyrefs. However, the work in this dissertation was already under-way and towards completion at that point.

tional and the object model do not support order constraints.

In our work, we thus chose to develop a common data model - *Sangam graph model*. The Sangam graph model is based on the common denominator of the existing data models - a graph, and can represent a subset of the common constraints present in existing data models. We base our model on the *schema intension graphs* (SIGs) from Miller *et al.* [MIR93]. The main design emphasis of the SIGs is to measure the information capacity (the intension [MIR93]) between different schemas of the same data model. While the SIGs are used to model schemas within the same data model, namely the relational model, we have found the SIGs to be sufficient to now also represent schemas from other data models such as the XML model. We term this simplified SIG the *Sangam Graph Model (SGM)*.

11.1 Sangam Graph Model

The Sangam Graph Model $\mathtt{M} = (\mathcal{N}, \mathcal{E})$ defines $\mathcal{N}$ a set of node types and $\mathcal{E}$ a set of edge types. We categorize the node types $\mathcal{N}$ of SGM as either complex (□) or *atomic* (◯); and the edge types $\mathcal{E}$ as either *containment* ($\rightarrow$) or *property* ($\dashrightarrow$). A *containment* edge is an edge between two complex nodes, while a *property* edge exists between a complex node and an atomic node. As the discussion here focuses on complex nodes and containment edges, we use the terms node and edge to imply these.

Sangam graph. The SGM allows us to model *Sangam graphs*. A Sangam graph $\mathtt{G} = (\mathrm{N}, \mathrm{E}, \lambda)$ is a directed graph where each node $\mathtt{n} \in \mathrm{N}$ is

either complex or atomic, i.e.,τ: n $\longrightarrow \mathcal{N}$; and each edge e $\in$ E is either a containment or a property edge, i.e.,τ':e $\longrightarrow \mathcal{E}$; and λ is a possibly infinite set of labels.

Types. Each complex node n $\in$ N represents a *user-defined type*, while each atomic node represents a *literal type* such as a *String* or an *integer*. We denote the set of user-defined types by Γ. Each node n is assigned a label l $\in \lambda$, i.e., τ":n$\longrightarrow$l. Any node with only outgoing property edges is termed a *leaf*.

NodeObjects. Each node n has associated with it a set of objects, termed *nodeObjects*. Each *nodeObject* o $\in$ I (n) is a pair <id, v> where id is a globally unique identifier and v is its data value. We term this set of nodeObjects the *extent* of the node n and denote this by I (n).

Relationships. The model permits the specification of binary relationships between nodes, represented by *directed* edges between two nodes. Each edge e $\in$ E is a labeled, directed edge between two nodes of N , with label l $\in \lambda$. An edge denoted as e:<n_1, n_2> indicates that e is an edge from node n_1 to node n_2. Each edge is annotated with a set of properties ζ, possibly empty. The set of annotations includes a *local order*, denoted by ρ. This gives the relative local ordering for all outgoing edges from a given node n in the Sangam graph. For example, given nodes n_1, n_2, n_3, and n_4, if e_1:<n_1, n_2>, e_2:<n_1, n_3>, e_3:<n_2, n_4> are three edges, then *local order* defines the position of the edge e_1 with respect to all other children[2] of

[2]We say that node n_j is a *child* of node n_i, if there exists an outgoing edge from n_i to n_j.

node n_1. In this example, $\rho(e_1) = 1$ and $\rho(e_2) = 2$. The local order of an edge e is specified as an integer, and the local order of sibling edges is continuous and differs by 1.

The *quantifier* annotation, denoted by Ω, associated with an edge e:$<n_1, n_2>$ specifies the subjectivity and injectivity of the binary relationship e between the nodes n_1 and n_2. A *quantifier* is given as a pair of integers [min:max], with $0 \leq min \leq max < \infty$ where min specifies the minimum and max the maximum occurrences of nodeObjects of a node n_2 for a given nodeObject of node n_1 associated via the binary relationship (edge) e.

EdgeObjects. Each edge e:$<n_1, n_2>$ has associated with it a set of *edgeObjects.* This set of *edgeObjects* is called the *extent* of the edge e and is denoted by R (e). Each edgeObject $o_e \in R$ is a triple $<id, o_1, o_2>$ where id is a system-generated identifier, nodeObject $o_1 \in I(n_1)$ and nodeObject $o_2 \in I(n_2)$. We denote the edgeObject using the edge notation, i.e., o_e:$<o_1, o_2>$. The edgeObject o_e denotes that nodeObject $o_1 \in I(n_1)$ is related to nodeObject $o_2 \in I(n_2)$. The set R may contain zero to multiple edges between the same two nodes.

Based on the above, we define a valid Sangam graph as follows.

Definition 6 (Valid Sangam graph) *A Sangam graph* $G = (N, E, \lambda)$ *is* **valid** *if for all nodes* $n \in N$, $\tau(n) \in \mathcal{N}$, *and for all edges* $e \in E$, $\tau'(e) \in \mathcal{E}$, *and the constraints (represented by the set of annotations* ζ *on an edge* e*) hold true by the extent of the edge* e, R(e)*.*

11.2 Converting Application Schemas and Data to Sangam graphs

A pre-requisite of our approach to achieve integration, is the ability to represent application schemas of different data models as Sangam graphs . The nodes in a Sangam graph represent the constructs of the underlying data model and the edges represent the relationships between the different constructs. Annotations on the edges capture the various constraints that may exist in the local data model. In this section we show how an XML schema can be translated into a Sangam graph and the XML documents corresponding to the XML schema can be placed as sets of objects (the extents) for the nodes and edges of the Sangam graph. We also show how the same can be done for relational schemas and data.

11.2.1 XML and Sangam graphs

Loading the XML Schema/DTD. The *loadXMLSchema* algorithm shown in Figures 11.1 and 11.2 is a two-pass algorithm that loads a given XML schema (or DTD) into a Sangam graph representation. Figure 11.1 gives the first pass while Figure 11.2 gives the second pass of the algorithm. In the first pass a Sangam node is created for every XML entity, *Element* or *Attribute*. The Sangam graph node is created by the method insertNode [3], which takes as parameters: (1) the label of the node - this is the name of the element or attribute; (2) the parent of the node - this is a Sangam graph node and represents the DTD for an element node and an element for an

[3]This operation is defined in Chapter 14

attribute node; (3) the edge label - the label for the binary relationship between the two nodes; (4) the quantifier - this is the quantifier annotation for the edge; and (5) order - the order annotation for the edge. If the parent node is specified as `null`, then the binary relationship, quantifier, and order must also be set to `null`. In general, an edge from the node that represents an element to the node representing an attribute denotes the binary relationship between the element and the attribute.

In the second pass, we load the content (definition) of elements. The relationship between two elements, given by the nesting of the elements, is denoted by a directed edge from the node `p` that represents the parent element to the node `n` that represents the child element. In XML, the definition of the child element is not inlined into the parent element. Rather the definition of the child element is a separate entity that can then be used by other parent elements. To model this in Sangam graphs, we introduce the notion of a *backpointer* edge. The *backpointer* edge is a containment edge that links a child node to its definition node. Thus, for every child node `n` a *backpointer* edge is added to its defining node `m`, where the node `m` represents the definition of the child element and is created during the first pass. This also enables us to handle recursion effectively in the Sangam graph.

An XML group is represented by a Sangam graph node and has an edge from the node representing the parent element to the child node representing the group. If there are multiple children nested in a parent element, the order of the children (be it a group or an element) is denoted by the order annotation on the edges from the parent node to the child. Similarly, the quantifier specifications "$*$" - child can occur zero or more times, "$+$" -

```
function loadXMLSchema (XMLDTD xmlS)
{
  DTDParser parser ← new DTDParser (xmlS , true )
  DTD dtd ← parser.parse
  // Create SAG Node for DTD
  SAGNode root ← createSAGNode(xmlS , NULL)
  //First Pass: Import the Root Elements and its Attributes
  Enumeration elemHash ← dtd.elements
  int Eorder ← 1
  while (elem.hasMoreElements()){
      DTDElement e ← (DTDElement)elem.nextElement()
      //Create Sangam Node for Element
      SAGNode elemNode ← createSAGNode(e.getName(), root, "elementEdge",
                      1, Eorder)
      Enumeration attrs ← e.attributes
      int Aorder ← 1
          // Traverse all Attributes
      while (attrs.hasMoreElements()){
          DTDAttribute a ← (DTDAttribute)attrs.nextElement()
          // Create Sangam Node for Attribute
          SAGNode attNode ← createSAGNode(a.getName(), elemNode,
                      "attributeEdge", 1, Aorder)
              Aorder++
          }
      Eorder++
      }

      //Second Pass: Load the content of the XML Elements
      (subelements, groups)
      Enumeration elem ← dtd.elements
      while (elem.hasMoreElements()){
          DTDElement e ← (DTDElement)elem.nextElement()
          importContent(e,root)
      }

  return root
}
```

Figure 11.1: The LoadXML Algorithm to Translate an XML DTD into a Sangam graph.

child can occur one or more times, "?" - child can occur zero or one times, and if none is specified then the child can occur exactly once, are translated into quantifier annotations on the edge from the parent to the child. The quantifier annotations are [0:n] for $*$, [1:n] for $+$, [0:1] for ?, and [1:1] otherwise. Currently, we cannot represent a group with a choice " | " in the Sangam graph. Figure 11.2 represents the second pass of the algorithm. The actual code for this can be downloaded from [Boo94].

Example 1 *Figure 11.3 shows the Sangam graph for the XMark benchmark schema of Figure 10.2. Here, each element and attribute is represented by a node. The edges between nodes, for example, the edge* e_1 *between the node labeled* `item` *and* `location` *represents the binary relationship between the* `item` *element and its sub-element* `location`*. The edge* e_1 *has an order annotation of* `1`*. The quantifier annotation [*`1:1`*] on edge* e_1 *denotes a functional edge, i.e., that there must be exactly one value of* `location` *that can participate in a binary relationship with one value of* `item`*. Similarly, the edge* e_2 *between the nodes with labels* `item` *and* `id` *represents the binary relationship between the XML element* `item` *and its attribute* `id`*. The order annotation for this edge is* `4`*. The order for attributes is not part of the XML model but is assigned as part of the Sangam graph. The information is discarded when converting the Sangam graph to an XML DTD or to another data model that does not keep track of order. The quantifier annotation [*`1:1`*] specifies a functional edge, i.e., there can only be one attribute* `id` *value per element* `item` *value.*

```
function importContent (DTDElement elem, SAGNode parent)
{       // Check content of the Element
   DTDITEM item ← elem.content()
   if item.getType() == (EMPTY || PCDATA)
       // return
   else {
       // Get quantifier and order
           Quantifier quantifier ← item.getQuantifier ()
           int order ← item.getOrder ()
           if (item.getType()) == GROUP {
               // Create group node - assign a default Name
               SAGNode gseNode ←
                           createSAGNode("gName", parent, "groupEdge",
                   quantifier, order)
           }
           elseif (item.getType()) == ELEMENT {
               //Create subelement node
               SAGNode gseNode ←
                       createSAGNode(item.getName(), parent,
                       "SubElementEdge", quantifier, order)
           }
           // Get all the items of a subelement or group
           Enumeration items = item.getItemsVec()
           for all it in items do: {
               importContent(it,gseNode)
           }
   }
}
```

Figure 11.2: The LoadXML Algorithm to Translate an XML DTD into a Sangam graph - The Second Pass.

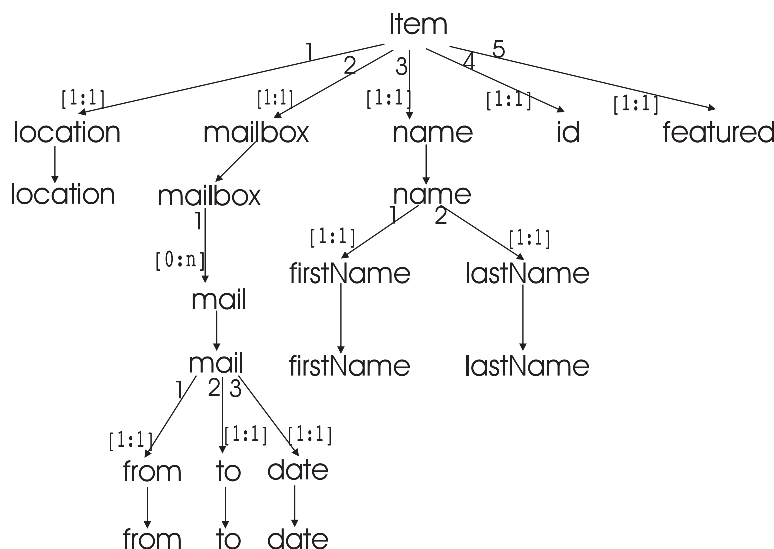


Figure 11.3: A Fragment of the XMark Benchmark DTD as shown in Figure 10.2 depicted as a Sangam Graph. No order and quantifier annotations are shown for the backpointer edges to distinguish them in the figure. These are defaulted to `1` and `[1:1] respectively.`

Loading the XML Data. While it is not necessary to materialize the Sangam graph, i.e., load the XML documents as objects of the Sangam graph nodes and edges, we present here for completeness one algorithm for extracting the XML documents and representing them as the extent of Sangam graph nodes and edges. The XML document is loaded into a `DOM` tree structure[4]. We traverse the DOM tree, root down, in a depth first manner. For each XML element, we first locate the matching Sangam graph node. The matching Sangam graph node is determined based on (1) ***name equality***, i.e., the element tag and the label of the Sangam graph node must be the same; and (2) the context, i.e., the element tags of all the elements in the path from the root to the element must be the same as the labels of all the Sangam graph nodes from the root to the matching node. This is similar to the Xpath identification. For every attribute the Sangam graph node match is done similarly but is much simpler as the context for an attribute is given directly by the parent element. Once a Sangam graph node match (called matching node) is found, then data in the XML document is translated as follows:

1. if it is an XML attribute, then a new object o_i is created with a unique identifier and the attribute value is translated as the value of the object.

2. if it is a non-**PCDATA** element or a group, an object o_i is created with a unique identifier and a null value.

3. if it is a **PCDATA** element, then a new object o_i is created with a

[4]We use `Java`'s XML parsers to do this.

unique identifier. The PCDATA value is stored as the value of the object.

In each of the above three cases, a new edgeObject o_e is created for the edge between the parent Sangam graph node (representing the element) and the current Sangam graph node [5]. The object is o_e represented as o_e: $<o_p, o_i>$, where object o_p is the parent object[6], and o_i is the child object.

Example 2 *Figure 11.4 shows an XML document conforming to the XMark DTD given in Figure 10.2. The XML fragment given in Figure 11.4 is mapped as the extent of the Sangam graph given in Figure 11.3 as follows: after the match for the element* `item` *is found, a new object* o_1 *is created with an identifier and a null value. The XML attribute* `id` *with value* `1` *is translated into an object* o_2 *for the node labeled* `id`*. The value of this object is set to* `1`*. A new object is also created with the object pair* o_e*:*$<o_1, o_2>$ *and is inserted into the extent of the edge between the nodes* `item` *and* `id`*. Similarly, the element* `location` *with the* ***PCDATA*** *value of* `'Paris'` *results in a new object* o_3 *whose value is set to* `'Paris'`*. A new object created with the value pair* $<o_1, o_3>$ *is inserted into the extent of the edge between the nodes with labels* `item` *and* `location`*. The element* `mailbox` *is a non-****PCDATA*** *element. Hence an object* o_4 *is created with a null value and is inserted into the extent of the node with label* `mailbox`*. The rest of the XML document is translated similarly. Figure 11.5 displays the Sangam graph with corresponding extents for each node and edge.*

[5]The edge itself is established during the load XML Schema process.

[6]Since we are doing a depth-first traversal, we always have a handle on the current parent object.

```
<?xml version="1.0" standalone="yes"? >

<item id="1", featured = 'no'>
    <location> Paris < /location>
    <mailbox> <mail mail='All set'>
        <from> Peter </from>
        <to> Pat </to>
        <date> Sept 23, 2001 < /date>
        < /mail>
    < /mailbox>
    <name><firstName>Peter< /firstName>
        <lastName> Unger < /lastName>
    < /name>
< /item>
```

Figure 11.4: A Fragment of the XMark Benchmark Document Conforming to the XMark Benchmark Schema in Figure 10.2.

Generating XML from a Sangam graph. The process of loading the XML DTD into a Sangam graph is *preserving*, i.e., the same XML DTD can now be generated from the Sangam graph by an inverse algorithm. Complex nodes in the Sangam graph represent the elements and the attributes. We further classify that a complex node with only atomic children nodes can be regarded as an XML attribute. All other nodes in the Sangam graph represent elements. Using this distinction of nodes, the XML DTD and document fragments can now be built by doing an in-order traversal of the Sangam graph starting at the root, and reversing the translation process done during the load time.

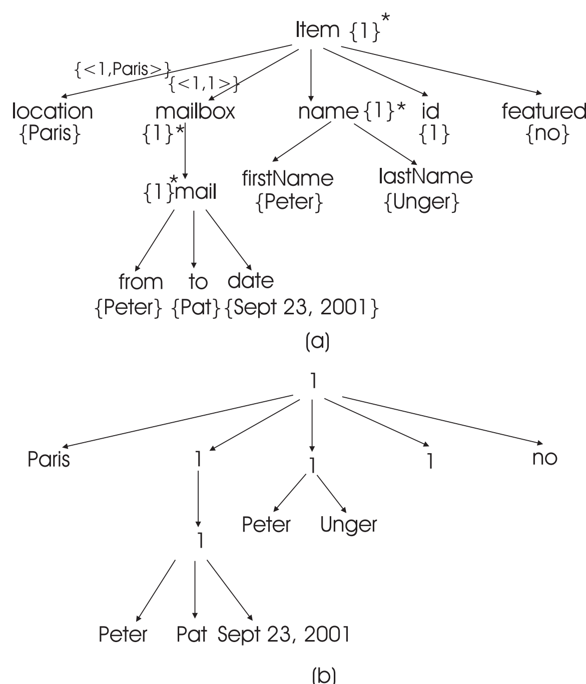


Figure 11.5: The Extent of the Sangam graph in Figure 11.3 based on the XML Document given in Figure 11.4. Part (a) depicts the extent. Here we show the extent of each node, and the extent of edge `e:<item, location>`. Part (b) presents just the object structure.

11.2.2 Relational Schemas and Sangam graphs

Loading the Relational Schema. Similar to the loading of an XML DTD, we first look at the algorithm for representing a relational schema as a Sangam graph. This algorithm is similar to the one given for the XML DTD in Figure 11.1. We give the main intuition based on an example. Figure 11.6 presents a relational schema with two tables and a foreign key constraint between the two tables. Each schema construct is first translated into a node in the Sangam graph. The schema constructs include a `table` and

a `column`. The domain type of each column is represented by an atomic node in the Sangam graph. For example, considering the relational schema in Figure 11.6 the table `ITEM` is represented by a node. Each of its columns, `location`, `mailBoxId`, `id`, `featured`, etc. is represented by complex nodes in the Sangam graph. The label of each node is the table or the column name respectively. The domain of a column is represented by atomic nodes. These are omitted from the figure to maintain its clarity.

```
ITEM(id VARCHAR(10) NOTNULL, featured VARCHAR(100),
      location VARCHAR(200), mailBoxID VARCHAR(10),
      lastName VARCHAR(200), firstName VARCHAR(200))

MAIL(from VARCHAR(50), to VARCHAR(50),
      date VARCHAR(8), mailBoxID INTEGER
      CONSTRAINT fk_id FOREIGN KEY (mailBoxID)
      REFERENCES ITEM(mailBoxID))
```

Figure 11.6: An Example Relational Schema.

All relationships, including the parent-child relationship between a table and its column, are represented by an edge from, for example, the node that represents the `ITEM` table to the node that represents the `id` column. As the relational model does not have order, the order annotation for each edge is defaulted to a serially increasing number during the schema load time. The quantifier annotation for an edge is decided based on the `NOTNULL` and the `UNIQUE` constraints on a column. If a column is set to `NOTNULL`, then the minimum quantifier on the edge between the parent (table node) and the node (column node) is set to `1`; else it is set to `0`. If a column is `UNIQUE`, then the maximum quantifier for the edge is set to `1`;

else it is set to ∞ to denote multiple occurrences.

Foreign key relationships between two tables are modeled by an edge between the node that represents the primary key column and the node that represents the foreign key column. The label of the foreign key edge is set to the name of the foreign key. In the example schema of Figure 11.6 this is `fk_id`. The order annotation is defaulted during the load, and the quantifier annotation is set to `[0:n]` to denote that one object of the primary key node may refer to zero or more objects of the foreign key node. Figure 11.7 represents the relational schema of Figure 11.6 as a Sangam graph.

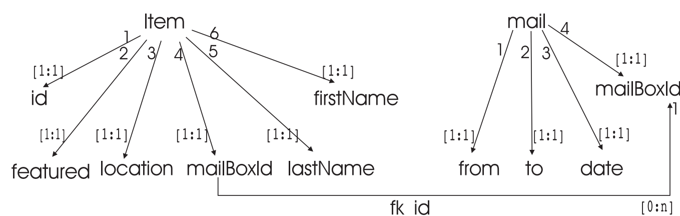


Figure 11.7: Relational Schema of Figure 11.6 depicted as a Sangam graph.

Loading the Relational Data. Again, the Sangam graph representation of the relational schema can be materialized if needed. Figure 11.1 depicts the relational tables `ITEM` and `MAIL` and their respective data. Similar to loading of XML documents, the relational data is read column-wise, i.e., all values of a particular column in a table. An object o with a unique object-identifier is generated for each row of the table. This object o is placed in the extent of the node that represents a table. For example, for the first row in the `ITEM` table, a new object is created and placed in the extent of the node `m` with label `ITEM`. Within the row, for each column a new object o_c is

created. The value of the object $\texttt{o}_c$ is set to the column's value. The object $\texttt{o}_c$ is then placed into the extent of the node `n` that represents the column. A new edge object $\texttt{o}_e$ is also created such that it now contains the pair $<\texttt{o}, \texttt{o}_c>$. This edge object is placed in the extent of the edge `e` between the nodes `m` and `n`. Values for other columns in the same row and consequently for the other rows are translated in a similar manner. Figure 11.5 gives the extent of the SAG obtained by translating the tables in Table 11.1.

id	featured	location	mailBoxID	lastName	firstName
1	no	Paris	1	Unger	Peter

from	to	date	mailBoxID
Peter	Pat	Sept 23, 2001	1

Table 11.1: Example Relational Tables `ITEM` and `MAIL` are Represented as Extent of the SAG as shown in Figure 11.5.

Generating Relational Schema and Data from a Sangam graph The loading process of the relational schema and data can be reversed to now translate the Sangam graph into a relational schema and the corresponding set of relational data. Similar to the generation of XML, for relational schema generation we distinguish Sangam graph nodes that represent relations versus those that represent attributes. An attribute is a complex node with only outgoing property edges. All other nodes represent relations. An edge between two nodes representing relations is translated to a foreign key construct. In addition to the reversing process to convert the Sangam graph into a relational schema, we also need to resolve two issues. One, a Sangam graph can handle set-valued attributes whereas the rela-

tional model cannot. We deal with this by building a separate relation and linking to the parent relation using a foreign key. The second is recursion. Recursion in a Sangam graph occurs when a cycle is formed when traversing a node with a *backpointer* edge. To handle recursion, each node with a *backpointer* edge that forms a cycle is translated to a new separate relation. Other than these distinctions, the process of translating a Sangam graph to a relational schema and data is a simple reverse of the loading algorithm.

11.3 Summary

In this chapter, we have described the data model utilized in our middle layer called the Sangam graph model. We have given algorithms to translate local application schemas in the XML and relational models into Sangam graphs, as well as algorithms to transform the corresponding data into Sangam graph extents.

The Sangam graph model can express the structure of schemas in a variety of data models such as the XML, relational, extended relational and object models. The Sangam graph constructs can express the common set of constraints in the relational and the XML model such as the key constraints, foreign key constraints, order and participation constraints. However, they cannot currently express constraints such as the functional dependency constraints.

One disadvantage of building a new data model is the lack of tools that exist for our unified data model - Sangam. This set of tools includes: a query language, a transformation language, parsers, design tools etc. These

tools come for ”free” if the unified data model is a high-level, commercial model such as the relational model or the XML model. However, as we will show in the next Chapter, there are many significant advantages for defining a low-level transformation language, and we believe that these compensate for some of the disadvantage.

Chapter 12

Cross Algebra

The purpose of our proposed cross algebra is to enable the transformation of an application schema and its associated data in one data model to an application schema and data in another model. To accomplish this the cross algebra operators transform a given input Sangam graph to an output Sangam graph. The cross algebra allows linear transformations [GY98] of the Sangam graph, i.e., nodes and edges of the input Sangam graph are mapped via the cross algebra operators to nodes or edges in the output Sangam graph. The linear transformations of a graph are the fundamental transformations, on the basis of which other more complex transformations can be defined [GY98]. The cross algebra operators include the `cross` operator denoted by $\otimes$ for the addition of a node to the output Sangam graph ; the `connect` operator denoted by $\ominus$ for the addition of a binary relationship between two nodes (an edge) in the output Sangam graph ; the `subdivide` denoted by $\circledless$ for splitting a binary relationship (an edge) in the input Sangam graph to a pair of binary relations (two edges) in the

output Sangam graph ; and the `smooth` operator denoted by $\oslash$, for converting two binary relations (two edges) in the input Sangam graph to one binary relation in the output Sangam graph. Deletion of nodes or edges can simply be accomplished by not mapping a given construct in the input Sangam graph to the output Sangam graph.

In this section, we detail the functionality of cross algebra operators. For each cross algebra operator we describe the transformation of the input Sangam graph (nodes, edges and set of objects associated with the nodes or edges) to the output Sangam graph (nodes, edges and set of objects respectively) that is accomplished by it; and preconditions, if any. We show how these algebra operators can be composed together to represent a algebra expression (Section 12.2). In Section 12.3.1, we present a binding of these operators to physical algebra operators and show how these can be evaluated. In the last section, Section 12.4 we give a complete example to show how basic inlining and shared inlining [STZ+99] techniques can be represented by these algebra operators, and hence show how XML can be transformed into relational schemas and data by a cross algebra expression.

12.1 Cross Algebra Operators

12.1.1 The Cross Operator

The `cross` algebra operator $\otimes$ corresponds to a node creation operation in the output Sangam graph. It takes as input a node `n` in the input Sangam graph and produces as output a node `n'` in the output Sangam graph. The cross operator is a total mapping, i.e., the objects in the extent of `n` given

by I (n) are mapped one-to-one to the objects in the extent of n′ given by I (n′) in the output Sangam graph.

Definition 7 (Cross) *Given a node* n *in the input Sangam graph* G*, a cross operator, denoted by* $\otimes$*, produces as output a node* n′ *in the output Sangam graph* G′*, i.e.,* $\otimes$: n $\longrightarrow$ n′ *such that for every* o = [v, id] $\in$ *I* (n)*, there exists an object* o′ = [v, id′] $\in$ *I* (n′) *such that the value* o.v = o′.v*; with I* (n) *and I* (n′) *the extents of nodes* n *and* n′ *respectively. We use* $\otimes_o$: o $\longrightarrow$ o′ *to represent this mapping between objects*[1]*;* l(n)=l(n′) *where* l(n) *and* l(n′) *are the labels of nodes* n *and* n′ *respectively; and* τ(n) = τ(n′)*, where* τ(n) *is the type of the node.*

Example 3 *Figure 12.1 (a) depicts a cross operator that maps the node* n *with label* featured *in the input Sangam graph* G *to the node* n′ *with label* featured′ *in the output Sangam graph* G′*. The extent of node* n *is mapped one-to-one to the extent of node* n′*. Hence, for all objects of the node* featured*, copies of the objects are created and placed in the extent of node* featured′*.*

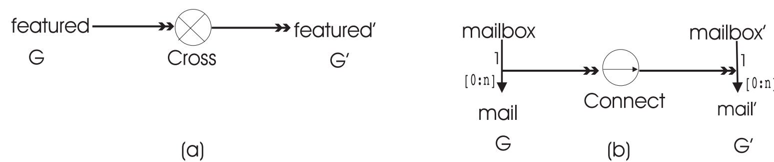


Figure 12.1: (a) The Cross Algebra Operator; (b) The Connect Algebra Operator.

[1]When it is clear from the context the subscript o is dropped from the $\otimes$.

12.1.2 The Connect Operator

A connect algebra operator ($\ominus$) corresponds to an edge creation in the output Sangam graph. It takes as input an edge e between two nodes n_1 and n_2 in the input Sangam graph G and produces an edge e' between two nodes n_1' and n_2' in the output Sangam graph G'. The connect operation succeeds if and only if nodes n_1 and n_2 have already been mapped to the nodes n_1' and n_2' respectively using two cross operators. The connect operator preserves the annotations of the edge e, i.e., the output edge e' will have the same quantifier and local ordering annotation as the input edge e.

Definition 8 (Connect Node) *Given an input Sangam graph* G *with two nodes* n_1 *and* n_2*, and edge* e:<n_1, n_2>*; and an output Sangam graph* G' *with nodes* n_1' *and* n_2' *such that* $\otimes(n_1) = n_1'$ *and* $\otimes(n_2) = n_2'$*. A connect operator* $\ominus$ *takes as input the edge* e:<n_1, n_2> $\in$ G *and produces as output edge* e'<n_1', n_2'> $\in$ G'*, i.e.,* $\ominus$:e$\longrightarrow$e' *such that for every object* o:<o_1, o_2> $\in$ *R* (e) *there is an object* o':<o_1', o_2'> $\in$ *R* (e') *such that* $o_1' = \otimes_o(o_1)$ *and* $o_2' = \otimes_o(o_2)$*;* ρ(e) $= \rho$(e')*;* Ω(e) $= \Omega$(e')*; the labels of* l(e) = l(e')*; and the types* τ(e) $= \tau$(e')*.*

Example 4 *Figure 12.1 (b) depicts a connect operator that maps the edge between the node* n_1 *with label* item *and the node* n_2 *with label* featured *to the edge between the nodes* n_1' *and* n_2' *with labels* ITEM *and* featured' *respectively in the output Sangam graph. Assuming that* $n_1' = \otimes(n_1)$*, and* $n_2' = \otimes(n_2)$*, then the objects of* n_1 *have been mapped to objects of* n_1'*. This also holds for the objects of* n_2 *and* n_2'*. The connect operator at the object level now builds, based on the relationship between the objects in the extents* I(n_1) *and* I(n_2)*, a relationship between the objects in extents* I(n_1') *and* I(n_2')*. Thus for example, the object* o_1 *of the node*

$\texttt{item}$ *is in relation with the object* o_2 *with value* $\texttt{true}$ *of the node* $\texttt{featured}$. *The connect operator will now build a relationship between the object* o_{11} *of the node* $\texttt{ITEM}$ *and the object* o_{22} *with value* $\texttt{true}$ *of the node* $\texttt{featured'}$. *This assumes that* $o_{11} = \otimes_o(o_1)$ *and* $o_{22} = \otimes_o(o_2)$*.*

12.1.3 The Smooth Operator

Let a Sangam graph $\texttt{G}$ have three nodes n_1, n_2, and n_3, and two binary relations represented by edges e_1:$< n_1, n_2 >$ and e_2:$< n_2, n_3 >$ as shown in Figure 12.2. The $\texttt{smooth}$ ($\circleddash$) operator replaces the binary relations represented by edges e_1 and e_2 in the Sangam graph $\texttt{G}$ with a new binary relation represented by edge e':$< n_1', n_3' >$ in the output Sangam graph $\texttt{G'}$. The smooth operator is successful only if $\otimes(n_1) = n_1'$ and $\otimes(n_3) = n_3'$. The *local order* annotation on the edge e' is set to the *local order* annotation of the edge e_1, i.e., $\rho(e') = \rho(e_1)$. However, the edge e' has a larger information capacity as it is a result of combining the capacities of relations represented by edges e_1 and e_2. Hence, the *quantifier* annotation of the edge e' is given as: $\Omega_{min}(e') = \Omega_{min}(e_1) * \Omega_{min}(e_2)$; and $\Omega_{max}(e') = \Omega_{max}(e_1) * \Omega_{max}(e_2)$. The $\texttt{smooth}$ operator produces a surjective edge e' if both e_1 and e_2 are surjective, or an injective edge e' if both e_1 and e_2 are injective.

Definition 9 (Smooth Operator) *Let* $\texttt{G}$ *be an input Sangam graph with three nodes* n_1, n_2, *and* n_3, *and edges* e_1:$< n_1, n_2 >$ *and* e_2:$< n_2, n_3 >$. *Let* $\texttt{G'}$ *be an output Sangam graph with nodes* n_1*'* *and* n_3*'* *such that* $\otimes(n_1) = n_1$*', and* $\otimes(n_3) = n_3$*'. A* $\texttt{smooth}$ *operator* $\circleddash$ *maps the edges* e_1:$<n_1, n_2>$ *and* e_2:$<n_2, n_3> \in \texttt{G}$ *to produce the edge* e':$< n_1', n_3' > \in \texttt{G'}$ *such that each pair of objects* o_{e1}:$<$

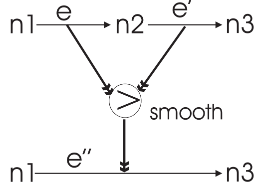


Figure 12.2: The Functionality of the Smooth Operator.

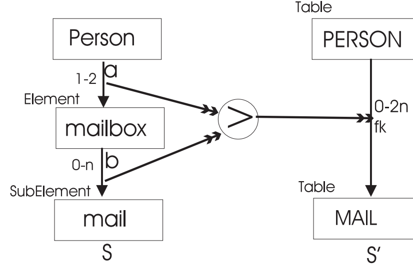


Figure 12.3: An Example of the Smooth Operator.

$o_1, o_2 >$ *and* o_{e2}:$< o_2, o_3 >$ *in R* (e_1) *and R* (e_2) *respectively, are represented by one edge object* o_{e3}:<o_1', o_3'> ∈ *R* (e'') *where* $o_1' = \otimes_o(o_1)$ *and* $o_3' = \otimes_o(o_3')$*;* $\rho(e') = \rho(e_1)$*; and* $\Omega_{min}(e') = \Omega_{min}(e_1) * \Omega_{min}(e_2)$*; and* $\Omega_{max}(e') = \Omega_{max}(e_1) * \Omega_{max}(e_2)$*; the labels of* e' *and* e_1 *are the same; and the types of* e' *and* e_1 *are the same, i.e.,* $\tau(e') = \tau(e_1)$*.*

Example 5 *Figure 12.3 depicts a* `smooth` *operator that maps the edges* e_1*:* `<Person, mailbox>`*, and* e_2*:* `<mailbox, mail>` *to the edge* e_3'*:* `<PERSON, MAIL>`*. Edge object* o_1*:* `<'Peter','Box1'>`[2] ∈ *R* (e_1) *and object* o_2*:* `<'Box1','All Done'>` ∈ *R* (e_2) *are mapped and a new object* o_3 *is created such that* o_3:`<'Peter','All Done'>`*. The object* o_3 ∈ *R* (e_3')*. The mapping of the objects is done as explained for the connect operator.*

12.1.4 The Subdivide Operator

A `subdivide` operator ⩹ performs the inverse operation of the `smooth` operator. Let `G` be a Sangam graph with two nodes n_1 and n_3 and edge e':$< n_1, n_3 >$ (Figure 12.4). The `subdivide` operator introduces a new node n_2'

[2]We use values here instead of oids for clarity of the example.

into the output Sangam graph $\texttt{G'}$ such that the edge $\texttt{e'}$ in $\texttt{G}$ is replaced by two edges $\texttt{e}_1$ and $\texttt{e}_2$ in the output Sangam graph $\texttt{G'}$: one between nodes $\texttt{n}_1\texttt{'}$ and $\texttt{n}_2\texttt{'}$ represented by edge $\texttt{e}_1$; and the other between nodes $\texttt{n}_2\texttt{'}$ and $\texttt{n}_3\texttt{'}$ represented by $\texttt{e}_2$. The `subdivide` operator is successful only if $\otimes(\texttt{n}_1) = \texttt{n}_1\texttt{'}$ and $\otimes(\texttt{n}_3) = \texttt{n}_3\texttt{'}$. The local order annotation for the edge $\texttt{e}_1$:$< \texttt{n}_1, \texttt{n}_2 >$ is the same as that for the edge $\texttt{e'}$, i.e., $\rho(\texttt{e}_1) = \rho(\texttt{e'})$ as they are incident from mapped nodes, i.e., $\otimes(\texttt{n}_1) = \texttt{n}_1\texttt{'}$. The edge $\texttt{e}_2$ is the only edge added for the node $\texttt{n}_2\texttt{'}$ and thus has a local order annotation of $\texttt{1}$, i.e., $\rho(\texttt{e}_2) = \texttt{1}$. During the subdivide operation, to preserve all the edges that exist between the extents $\texttt{I}(\texttt{n}_1)$ and $\texttt{I}(\texttt{n}_3)$ of nodes $\texttt{n}_1$ and $\texttt{n}_3$ respectively, the output edges $\texttt{e}_1$ and $\texttt{e}_2$ are assigned quantifier annotations as follows. If $\Omega_{min}(\texttt{e'}) = \texttt{0}$, then the quantifier range for $\texttt{e}_1$, $\Omega(\texttt{e}_1) = [0:1]$, else $\Omega(\texttt{e}_1) = [1:1]$. The quantifier of edge $\texttt{e}_2$, $\Omega(\texttt{e}_2) = \Omega(\texttt{e'})$. Thus the two edges $\texttt{e}_1$ and $\texttt{e}_2$ model a surjective relationship if $\texttt{e'}$ models a surjective relationship.

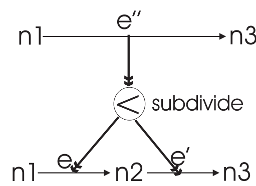


Figure 12.4: The Functionality of the Subdivide Node.

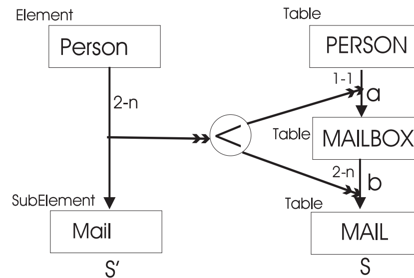


Figure 12.5: An Example of the Subdivide Node.

Definition 10 (Subdivide Operator) *Let* $\texttt{G}$ *be an input Sangam graph with two nodes* $\texttt{n}_1$ *and* $\texttt{n}_3$ *and an edge* $\texttt{e'}$. *Let* $\texttt{G'}$ *be the output Sangam graph* $\texttt{G'}$ *with nodes* $\texttt{n}_1$*',* $\texttt{n}_2$*' and* $\texttt{n}_3$*' such that* $\otimes(\texttt{n}_1) = \texttt{n}_1$*', and* $\otimes(\texttt{n}_3) = \texttt{n}_3$*'. A subdivide*

operator $\oslash$ takes as input the edge e':<n_1, n_3> $\in$ G, *and produces as output one node* n_2' *and two edges* e_1:<n_1', n_2'> *and* e_2:<n_2', n_3'> $\in$ G', *such that (1) each edge object* o_{e1}:<o_1, o_3> $\in$ *R* (e) *is mapped to edge object* o'_{e1}:<o_1', o_{n2}> $\in$ *R* (e_1), *and edge object* o_{e2}':<o_{n2}, o_3'> $\in$ *R* (e_2) *where* $o_1' = \otimes_o(o_1)$, $o_3' = \otimes_o(o_3)$, *and* o_{n2} *is a new node object that is created such that* $o_{n2} \in I\ (n_2')$; *(2)* $\Omega_{min}(e_1) = 0$, *if* $\Omega_{min}(e') = 0$, *and* $\Omega_{min}(e_1) = 1$ *otherwise;* $\Omega_{max}(e) = 1$; $\Omega(e_2) = \Omega(e')$; $\rho(e_1) = \rho(e')$ *and* $\rho(e_2) = 1$; *the labels of* e_1 *and* e_2 *are set to be the same as the label of* e'; *the types of* e_1 *and* e_2 *are set to be the same as the type of* e'; *the label of the new node* n_2' *is set to a system-defined label; and* $\tau(n_2') = \tau(n_1)$.

Example 6 *Figure 12.5 depicts the inverse of the smooth operator example from Figure 12.3. It divides the edge* e_3':<PERSON, MAIL> *into two edges namely,* e_1:<Person, MailBox>, *and* e_2:<MailBox, Mail>. *Here an edge object* o_e:<'Peter', 'All Done'>, *relates the object* o_1[3] $\in$ *I* (PERSON) *and the object* $o_3 \in$ *I* (MAIL) *in the input Sangam graph. A new object* o_2' *is created and inserted into the extent of the node* MailBox. *A new edge object* o_{e1} *is created such that* o_{e1}:<o_1', o_2'> *with* $o_1' = \otimes_o(o_1)$. *A new edge object* o_{e2} *is created such that* o_{e2}:<o_2', o_3'> *with* $o_3' = \otimes_o(o_3)$. *The edge object* o_{e1} *is inserted into the extent R* (e_1) *and the edge object* e_{e2} *is inserted into the extent R* (e_2). *Hence, all objects that were connected by one edge are now connected by a path of two edges.*

[3] This object represents the object with value Peter.

12.1.5 Notation

We use the notation given in Table 12.1 to indicate the cross algebra operators. In general, $\mathtt{op}_{out}$(*in*) represents an algebra operator that operates on input `in`, where `in` is either a node or an edge (or two edges for the smooth operator) in a Sangam graph, and produces the output `out` which again is either a node or an edge (or a node and two edges for the sub-divide operator) in a Sangam graph. These are specified using the labels assigned to the nodes or edges in a Sangam graph.

Notation	Description
$\otimes_{n'}$ (n)	Cross operator with input n and output n'
$\ominus_{e'}$ (e)	Connect operator maps edge e:<n_1, n_2> to edge e':<n_1', n_2'>
$\ominus_{e'}$(e_1, e_2)	Smooth node maps edges e_1:<n_1, n_2> and e_2:<n_2, n_3> to edge e':<n_1', n_3'> in the output
$\ominus_{e1',e2',n2'}$(e)	Subdivide node maps edge e:<n_1, n_3> to e1':<n_1', n_2'> and e2':<n_2', n_3'> in the output

Table 12.1: Notation Used for Cross Algebra Operators.

12.2 Cross Algebra Trees

In relational algebra, algebra operators are nested to form one algebra expression. These operators are combined together via *derivation* by applying one operator(s) to the result of one or more operators. The relational algebra has the advantage of operating within the relational model, i.e., its

input and output are both relational. Moreover, each relational algebra operator operates on complete relations, treating a relation as the basic unit for input and output. Attributes of a relation for example can only be mapped (or not mapped, in the case of projection) into attributes of the output relation (and only if the relation itself is mapped). In other words, the schema of the output is implied by the semantics of the relational operators, i.e., each operator has a built-in template of the output schema. Similarly, XML algebra operates on the granularity of an ordered collection of XML fragments, where each fragment is a complex nested XML element and produces as output also an ordered collection of XML fragments [FFM+01a].

Cross algebra, however, operates at the granularity of individual nodes and edges in a Sangam graph. A cross operator takes a node as input and produces a node as output, whereas the connect, smooth and subdivide operators operate on edge(s) in the Sangam graph and produce edge(s) as output (possibly adding or removing nodes). However, the difference between the cross algebra, and the relational and XML algebras, lies in the fact that a node in a Sangam graph may represent data model specific constructs such as an XML element, or a relational table, or an XML/relational attribute. Correspondingly, an edge may, for example, represent the relationship between an XML element and its attribute, or a foreign key between two relations. An algebra that goes crosses different data models must therefore be flexible in order to accommodate the variance in the sizes of *modeling units* of each individual data model. For example, the modeling unit for the relational model is two-level deep, i.e. relations and their columns, whereas for XML it is `n`-level deep and at times recursive thereby allowing

nested element structures. As we cannot predict the right granularity to accommodate all data models, we use the smallest granularity within the Sangam graph as a modeling unit. This has the advantage of allowing the mapping of constructs and relationships in one data model to constructs and relationships in another data model. However, given this low-level of granularity we now need additional mechanisms to express the mapping of arbitrary complete units such as a relation or a complex nested element from the input graph to the output graph. To enable this, we introduce a new type of composition of cross algebra operators that allows several algebra operators to collaborate and jointly operate on different parts of the input Sangam graph, and together produce one (connected) Sangam graph. We term this a *context dependency composition.* In this section, we give details of the more traditional derivation composition of the cross algebra operators, the context dependency composition, and the combination of the two compositions into one valid algebra expression.

12.2.1 Derivation Trees

Figure 12.6 gives an example of a derivation composition that transforms the path in the Sangam graph shown in Figure 12.6 (a) to the edge in the Sangam graph given in Figure 12.6 (c) by applying three smooth nodes ⧁. Let e_1:<A,B>, e_2:<B, C>, e_3:<C, D> and e_4:<D, E> be edges in the input Sangam graph G. Operators $op1_{e1'}(e_1, e_2)$ and $op2_{e2'}(e_3, e_4)$ are applied to the input edges e_1 and e_2, and e_3 and e_4 respectively to first produce the intermediate edges e_1':<A', C'> and e_2'<C', E'> as shown in Fig-

ure 12.6 (b)[4]. The operator $\texttt{op3}_{e3''}(\texttt{e}_1\texttt{'}, \texttt{e}_2\texttt{'})$ operates on these intermediate edges $\texttt{e}_1\texttt{'}$ and $\texttt{e}_2$' and produces the desired output edge $\texttt{e}_3\texttt{''}$:<A'',E''> as shown in Figure 12.6 (c). One approach to achieving this is to first produce the intermediate edges and then the final output edge $\texttt{e}_3$". Another option is to nest the operators. Thus, the output of the algebra expression $\texttt{op3}_{e3''}(\texttt{op1}_{e1'}(\texttt{e}_1, \texttt{e}_2), \texttt{op2}_{e2'}(\texttt{e}_3, \texttt{e}_4))$ is the output edge $\texttt{e}_3$". The output of the operator `op3` is said to be ***derived*** from the outputs of operators `op1` and `op2`, or put differently, the output of operators `op1` and `op2` are the ***inputs*** of operator `op3` and are consumed by `op3`.

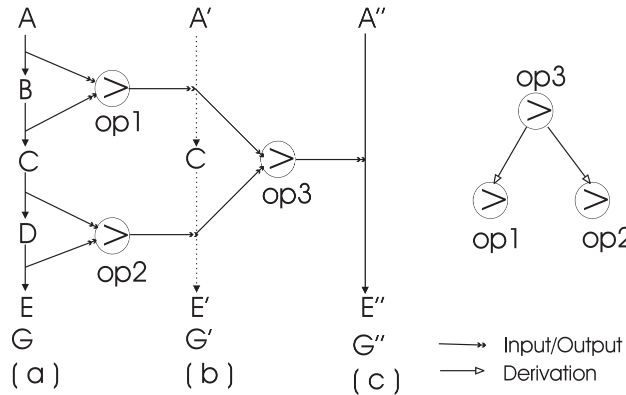


Figure 12.6: Derivation Composition.

We use the term *derivation expression* to represent the nesting of algebra operators. We define a derivation expression $\texttt{DT}_o$ as follows.

Definition 11 *Given an input Sangam graph* `G`*, a derivation expression* $\texttt{DT}_i$ *is given as:*

[4]We do not include here the cross operators needed to map the nodes of the graph. We defer that part of the discussion to Section 12.2.2.

$$\mathtt{DT}_{\mathtt{o}(\mathtt{out_o})}(in_o) = \begin{cases} op_{i(out_i)}(in_i) & \textit{for a single operation } op_i \in \{\otimes, \ominus, \ogreaterthan, \olessthan\} \textit{ with } \mathtt{op}_i \textit{ semantics as defined in Definitions 7, 8, 9 and 10} \\ op_{i(out_i)}(\mathtt{DT}_{\mathtt{j}(\mathtt{out_j})}(\mathtt{in_j})) & \textit{where } \mathtt{op}_i \in \{\otimes, \ominus, \olessthan\} \textit{ and } DT_j \textit{ a derivation tree and } \mathtt{in}_i = \mathtt{out}_j \\ op_{i(out_i)}((\mathtt{DT}_{\mathtt{k}(\mathtt{out_k})}(\mathtt{in_k})), (\mathtt{DT}_{\mathtt{l}(\mathtt{out_l})}(\mathtt{in_l}))) & \textit{where } op_i \in \{\ogreaterthan\}, \textit{ and } DT_k \textit{ and } DT_l \textit{ are derivation trees and } \mathtt{in}_i = \{\mathtt{out}_k, \mathtt{out}_l\} \end{cases}$$

produces as output $\mathtt{out}_o$ ***node and edge elements for an output Sangam graph*** $\mathtt{G'}$***, such that*** $\mathtt{out}_o$ ***is the output of the root operator*** $\mathtt{op}_i$ ***and thus also of the complete derivation tree*** $\mathtt{DT}_o$***.***

Here $\mathtt{out}_o$ denotes the output of the derivation tree $\mathtt{DT}_o$, that is Sangam graph nodes or edges generated by $\mathtt{DT}_o$; and $\mathtt{in}_o$ denotes the input, i.e., Sangam graph nodes and/or edges, the derivation tree operates on. The output of the derivation tree $\mathtt{DT}_o$ is equal to the output $\mathtt{out}_i$, i.e., the output produced by the top-most algebra operator $\mathtt{op}_i$ of the derivation

tree. When $\mathtt{DT}_o$ is a derivation tree with more than one algebra operator, then the output of the outermost $\mathtt{op}_i$ is calculated by using as input the outputs generated by its respective inner nested derivation trees $\mathtt{DT}_j$ or ($\mathtt{DT}_k$ and $\mathtt{DT}_l$). Here the nested derivation trees generate the inputs of the operator $\mathtt{op}_i$.

We use "(" and ")" pairs to denote nesting, i.e., derivation of $\mathtt{op}_i$ from the derivation tree $\mathtt{DT}_j$. We use the symbol "," to separate input arguments of an operator $\mathtt{op}_i$.

A derivation expression is graphically represented as a *derivation tree.*

Definition 12 *Given a derivation expression $\mathtt{DT}_o$ as defined in Definition 11, $\mathtt{T}_D$ = ($\mathtt{N}_o$, $\mathtt{E}_d$) is a derivation tree such that (1) each node $\mathtt{n}_i \in \mathtt{N}_o$ represents a cross algebra operator ($\otimes$, $\ominus$, $\oslash$, $\owedge$) in $\mathtt{DT}_o$; and (2) each edge $\mathtt{e}_i \in \mathtt{E}_d$ represents the nesting "(" and ")" in $\mathtt{DT}_o$, as a derivation edge $\multimap$ from the parent (outer nested) operator $\mathtt{op}_i$ to each of the children (inner nested) operators. If any child (inner nested) operator itself is a derivation expression $\mathtt{DT}_j$, then a derivation edge $\multimap$ exists from the parent (outer nested) operator $\mathtt{op}_i$ to the root operator of the internal derivation tree $\mathtt{DT}_j$.*

Henceforth, we use the terms derivation expression and derivation tree interchangeably and distinguish between them only when needed.

Consider the derivation tree in Figure 12.6. The expression for this tree is given as:

$$\mathtt{DT1} = \mathtt{op3}_{e3''}(\mathtt{op1}_{e1'}(\mathtt{e}_1, \mathtt{e}_2), \mathtt{op2}_{e2'}(\mathtt{e}_3, \mathtt{e}_4)) \tag{12.1}$$

Graphically, this is represented by two derivation edges from the parent operator op3 to the children operators op1 and op2, given as op3⇾op1, and op3⇾op2. Note that a derivation tree alone cannot be guaranteed to produce a valid output. For example, op3 here produces an edge e_3'' without any end nodes. This edge alone is not a valid Sangam graph. In the next few sections, we discuss how a derivation tree can be used to indeed produce valid output.

Lemma 4 *A derivation tree* T_D *as defined in Definition 12 and given by the expression* $DT_{o(out_o)}(in_o)$ *is a rooted directed acyclic graph (DAG).*

Proof: This can be proven by a simple proof by induction.
Base Case - Let $DT_{o(out_o)}(in_o) = op_{i(out_i)}(in_i)$. As the DT is comprised of one operator only, clearly it is acyclic and there is only one root.
Hypothesis - Let:

$$DT_{k(out_k)}(in_k) = op_{k(out_k)}(DT_{j(out_j)}(in_j)) \tag{12.2}$$

such that DT_k is a derivation tree with k operators. Let us assume that DT_k with k operators is acyclic, (i.e., that all edges (derivation edges) from the root operator op_k are in the same direction from the root to its children (leaves of the tree)), and rooted.
Induction k← k+1 - Let us assume that we have a derivation tree DT_k with k operators. We can now construct a derivation tree DT_{k+1} with k+1 operators in the following three ways:

Case 1: Operator $\mathtt{op}_{k+1}$ is added as the new root of $\mathtt{DT}_k$. The resulting derivation tree $\mathtt{DT}_{k+1}$ is a DAG.

Let $\mathtt{op}_{k+1}$ be the root operator of $\mathtt{DT}_{k+1}$ such that:

$$\mathtt{DT}_{\mathtt{k+1}(\mathtt{out}_{\mathtt{k+1}})}(\mathtt{in}_{\mathtt{k+1}}) = \mathtt{op}_{\mathtt{k+1}(\mathtt{out}_{\mathtt{k+1}})}(\mathtt{DT}_{\mathtt{k}}). \tag{12.3}$$

Re-writing Equation 12.3 using Equation 12.2, we get:

$$\mathtt{DT}_{\mathtt{k+1}(\mathtt{out}_{\mathtt{k+1}})}(\mathtt{in}_{\mathtt{k+1}}) = \mathtt{op}_{\mathtt{k+1}(\mathtt{out}_{\mathtt{k+1}})}(\mathtt{op}_{\mathtt{k}(\mathtt{out}_{\mathtt{k}})}(\mathtt{DT}_{\mathtt{j}(\mathtt{out}_{\mathtt{j}})}(\mathtt{in}_{\mathtt{j}}))) \tag{12.4}$$

That is, a single derivation edge exists from the root operator $\mathtt{op}_{k+1}$ to its direct child operator $\mathtt{op}_k$ (Definition 12). We know by our hypothesis that a derivation tree $\mathtt{DT}_k$ given as $\mathtt{DT}_k = \mathtt{op}_{k(out_k)}(\mathtt{DT}_{j(out_j)}(\mathtt{in}_j))$ (Equation 12.2) with $\mathtt{k}$ operators is acyclic, and all its edges are from the root $\mathtt{op}_k$ to its children. Since $\mathtt{DT}_{k+1}$ has $\mathtt{k} + \mathtt{1}$ operators with a derivation edge from the root $\mathtt{op}_{k+1}$ to $\mathtt{op}_k$, all the edges are in the same direction, i.e., from the root to the children. Hence, $\mathtt{DT}_{k+1}$ is acyclic as well.

Case 2: Operator $\mathtt{op}_{k+1}$ is added as the new leaf of $\mathtt{DT}_k$. The resulting derivation tree $\mathtt{DT}_{k+1}$ is a DAG.

Case 3: Replace existing subtree $\mathtt{DT}_i$ with a new tree $\mathtt{DT}_{i+1}$ with $\mathtt{i} + \mathtt{1}$ operators. We know that DT_{i+1} is a DAG, as $\mathtt{i} < \mathtt{k}$. Hence, the new derivation tree $\mathtt{DT}_{k+1}$ is also a DAG. □

12.2.2 Context Dependency Trees

While a derivation tree may consume multiple edges and nodes as input, it always produces exactly one edge or one node, or in the case of the subdivide root operator one node and two edges as defined above (Definition 11). It never produces an entire output Sangam graph. Nor can complete output Sangam graphs be produced by individual operators that are not nested. Consider the three cross algebra operators `op1` (⊗), `op2` (⊗) and `op3` (⧁) that map the node `A` to `A'`, `B` to `B'` and the edge `e:<A, B>` to edge `e':<A', B'>` respectively. Without a mechanism to represent a grouping of these operators, it is not possible to indicate (1) that the three operators together work on the same input Sangam graph and produce one single output Sangam graph `G'` composed of two nodes `A'` and `B'`, and an edge `e':<A,B>` as shown in Figure 12.7; (2) the operator `op3` can be successfully evaluated only after the evaluation of `op2` and `op1`; and (3) output of `op1` and `op2` is *used* by `op3` but not consumed by it.

To remedy this, we introduce a ***context dependency tree*** to express the above semantics. A context dependency tree `CT` represents a rooted, hierarchical grouping of cross algebra operators that together map an input Sangam graph `G` to an output Sangam graph `G'`. Each operator `op` in the context dependency tree `CT` operates on individual nodes or edges in the input Sangam graph `G` and maps them to individual nodes or edges in the same output Sangam graph `G'`.

Figure 12.7 denotes such a context dependency tree `CT` composed of three cross algebra operators. Here, the algebra operators $\mathtt{op1}_{\mathtt{A'}}(\mathtt{A})$ and

$\texttt{op2}_{\texttt{B}'}$(B) are cross operators that map the nodes A and B in G to nodes A′ and B' respectively in the output Sangam graph G′. The algebra operator $\texttt{op3}_{\texttt{e}'}$(e), a ⊖ operator is the root of CT and maps the edge e:<A, B> between the nodes A and B in the input Sangam graph G to the edge e′:<A′, B′> between the nodes A′ and B′ in the output Sangam graph G′. Here the outputs of all operators op1, op2, and op3 together produce the output Sangam graph G′.

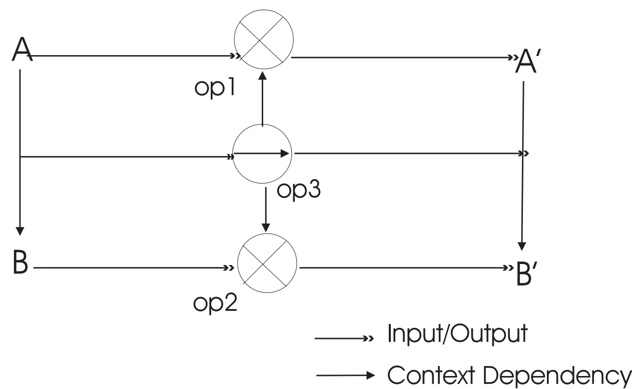


Figure 12.7: Context Dependency Composition Example.

This grouping of algebra operators, called *context dependency tree*, is denoted by $\texttt{CT}_o$ and is defined as follows.

Definition 13 *Given an input Sangam graph* G*, a context dependency tree* $\texttt{CT}_o$

specified as:

$$\mathtt{CT}_{o(out_o)}(\mathtt{in}_o) = \begin{cases} op_{i(out_i)}(in_i) & \text{for a single operation } op_i \in \{\otimes, \ominus, \oslash, \circledcirc\}, \text{ and } \mathtt{out}_o = \mathtt{out}_i \\ op_{i(out_i)}(in_i), (CT_{k(out_k)}(in_k)) [\circ\ (CT_{l(out_l)}(in_l))]^{+}) & \text{where } op_i \text{ is the parent operator of } CT_k \text{ and } CT_l \text{ denoting that } op_i \text{ must be executed after } CT_k \text{ and } CT_l. \ op_i \text{ uses outputs, inputs and mapping of } CT_k \text{ and } CT_l \text{ and } out_o = out_j \bigcup out_k \bigcup out_l. \text{ The symbol } []^{+} \text{ is part of the grammar syntax and is defined below.} \end{cases}$$

operates on nodes $\mathtt{n}_i$ *and edges* $\mathtt{e}_i \in \mathtt{G}$*, and produces as output a Sangam graph* $\mathtt{G'}$ *such that all nodes* $\mathtt{n}_i$*' and/or edges* $\mathtt{e}_i$*' produced as output by any of the individual operators* $\mathtt{op}_i \in \mathtt{CT}$ *are in* $\mathtt{G'}$*.*

Here $\mathtt{in}_o$ denotes the input of the context dependency tree $\mathtt{CT}_o$, that is Sangam graph nodes or edges of graph $\mathtt{G}$. The expression $\mathtt{out}_o$ denotes the final output of the context dependency tree $\mathtt{CT}_o$. The output $\mathtt{out}_o$ is the union of the outputs of all operators that compose the context dependency

tree. Thus, in Definition 13, $\texttt{out}_o = \texttt{out}_i \bigcup \texttt{out}_k \bigcup \texttt{out}_l$, to denote that the final output is the output of the operator $\texttt{op}_i$ and the context dependency trees $\texttt{CT}_k$ and $\texttt{CT}_l$.

In Definition 13 the symbol "$+$" is part of the BNF grammar syntax to indicate that the expression contained in "[ ]" may occur one or more times.

Definition 14 *Given a context dependency expression $\texttt{CT}_o$ as defined in Definition 13, $\texttt{T}_C = (\texttt{N}_o, \texttt{E}_c)$ is a context dependency tree such that (1) each node $\texttt{n}_i \in \texttt{N}_o$ represents a cross algebra operator ($\otimes$, $\ominus$, $\circleddash$, $\oslash$) in the expression $\texttt{CT}_o$; and (2) each edge $\texttt{e}_i \in \texttt{E}_d$ is a context dependency edge $\rightarrow$ from the parent (left) operator to the child (right) operator, and represents the symbol "," in the expression $\texttt{CT}_o$. The symbol "$\circ$" in the expression $\texttt{CT}_o$ is denoted by fact that the operators (or trees) share a common parent (left) operator. No edges exist between the sibling operators. If a parent (left) operator has more than one child, where the children (right) operators are separated by $\circ$, then a context dependency edge exists from the parent (left) to each of the children (right) operators. If a child (right) operator is a context dependency tree, then the context dependency edge $\rightarrow$ exists from the parent operator $\texttt{op}_i$ to the root operator $\texttt{op}_j$ of the context dependency tree.*

Henceforth, we use the terms context dependency expression and context dependency trees interchangeably and distinguish between them only when necessary.

Thus, the expression for the context dependency tree in Figure 12.7 may be given as:

$$\mathtt{CT}_{\mathtt{out}}(\mathtt{in}) = \mathtt{op3}_{e'}(\mathtt{e}), (\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op2}_{\mathtt{B'}}(\mathtt{B})). \quad (12.5)$$

Here the operator op3 is the root of the context tree, and op1 and op2 are the children operators. Here e:<A, B> and e':<A', B'>. The final output $\mathtt{out} = \mathtt{e'} \bigcup \mathtt{A'} \bigcup \mathtt{B'}$ corresponds to the final output SAG G'.

The context dependency expression $\mathtt{CT}_o$ is evaluated from right to left and from inside out. That is all children operators are evaluated prior to the evaluation of their parent operator. The order of evaluation between the sibling operations is immaterial.

Beyond the order of evaluation, the context dependency relation between two operators op3→op1 (Figure 12.7) implies that the operator op3 uses the following three pieces of information in its calculation: (1) the input of op1; (2) the output of op1; and (3) the mapping ϕ of op1 as established by the type $\otimes$, $\ominus$, $\circledcirc$ and $\circledcirc$ of op1 given in Definitions 7, 8, 9, and 10.

Lemma 5 *A context dependency tree* $\mathtt{T}_C$ *as defined in Definition 14 and given by the expression* $\mathtt{CT}_{l(out_l)}(\mathtt{in}_l)$ *(Definition 13) is a rooted, directed acyclic graph.*

Proof: This can be proven by a simple proof by induction.

Base Case - Let $\mathtt{CT}_{o(out_o)}(\mathtt{in}_o) = \mathtt{op}_{i(out_i)}(\mathtt{in}_i)$. As the $\mathtt{CT}_o$ is comprised of one operator only, clearly it is acyclic.

Hypothesis - Let:

$$\mathtt{CT}_{\mathtt{k(out_k)}}(\mathtt{in_k}) = \mathtt{op}_{\mathtt{k(out_k)}}(\mathtt{in_k}), (\mathtt{CT}_{\mathtt{j(out_j)}}(\mathtt{in_j})) \tag{12.6}$$

be a context dependency tree such that $\mathtt{CT}_k$ is a context dependency tree with $\mathtt{k}$ operators. Let us assume that $\mathtt{CT}_k$ with $\mathtt{k}$ operators is acyclic, and all edges (context dependency edges) are in the same direction, i.e., from the root $\mathtt{op}_k$ to the children (leaves).

Induction -

Case 1: Operator $\mathtt{op}_{k+1}$ is added as the new root of $\mathtt{CT}_k$.

Given any $\mathtt{CT}_{k+1}$, by Definition 13 we have:

$$\mathtt{CT}_{\mathtt{k+1(out_{k+1})}}(\mathtt{in_{k+1}}) = \mathtt{op}_{\mathtt{k+1(out_{k+1})}}(\mathtt{in_{k+1}}), (\mathtt{CT}_{\mathtt{k(out_k)}}(\mathtt{in_k})) \tag{12.7}$$

Re-writing this expression using Equation 12.6 we have:

$$\mathtt{CT}_{\mathtt{k+1(out_{k+1})}}(\mathtt{in_{k+1}}) = \mathtt{op}_{\mathtt{k+1(out_{k+1})}}(\mathtt{in_{k+1}}), (\mathtt{op}_{\mathtt{k(out_k)}}(\mathtt{in_k}), (\mathtt{CT}_{\mathtt{j(out_j)}}(\mathtt{in_j}))) \tag{12.8}$$

That is a single context dependency edge exists from the parent (left) operator $\mathtt{op}_{k+1}$ to the child operator $\mathtt{op}_k$. We know by our hypothesis that a context dependency tree given as: $\mathtt{CT}_k = (\mathtt{op}_{k(out_k)}, (\mathtt{CT}_{j(out_j)}(\mathtt{in}_j)))$ (Equation 12.6) is acyclic, and all its edges are from the parent $\mathtt{op}_k$ to the

children (leaves) operators. Since $\mathtt{CT}_{k+1}$ has $\mathtt{k\ +\ 1}$ operators with a context dependency edge from the parent $\mathtt{op}_{k+1}$ to $\mathtt{op}_k$, all edges are in the same direction. Hence, $\mathtt{CT}_{k+1}$ is acyclic.

Case 2: Operator $\mathtt{op}_{k+1}$ is added as the new leaf of $\mathtt{CT}_k$.

Given any $\mathtt{CT}_{k+1}$, by Definition 13 we can have:

$$\mathtt{CT}_{\mathtt{k+1(out_{k+1})}}(\mathtt{in_{k+1}}) = \mathtt{op}_{\mathtt{k(out_k)}}(\mathtt{in_k}), (\mathtt{CT}_{\mathtt{j(out_j)}}(\mathtt{in_j}) \circ \mathtt{op}_{\mathtt{k+1(out_{k+1})}}(\mathtt{in_{k+1}})) \tag{12.9}$$

That is the operator $\mathtt{op}_{k+1}$ is a child operator of $\mathtt{op}_k$, and a single context dependency edge exists from the parent (left) operator $\mathtt{op}_k$ to the child operator $\mathtt{op}_{k+1}$ (Definition 14). We know by our hypothesis that a context dependency tree given as: $\mathtt{CT}_k = (\mathtt{op}_{k(out_k)}, (\mathtt{CT}_{j(out_j)}(\mathtt{in}_j)))$ (Equation 12.6) is acyclic, and all its edges are from the parent $\mathtt{op}_k$ to the children (leaves) operators. Since $\mathtt{CT}_{k+1}$ has $\mathtt{k\ +\ 1}$ operators with a context dependency edge from the parent $\mathtt{op}_k$ to $\mathtt{op}_{k+1}$, all edges are in the same direction. Hence, $\mathtt{CT}_{k+1}$ is acyclic.

Case 3: Some tree $\mathtt{CT}_i$ is replaced by a new tree $\mathtt{CT}_{i+1}$. As $\mathtt{i} < \mathtt{k}$, we know by our assumption that $\mathtt{CT}_{i+1}$ is a DAG. □

12.2.3 Cross Algebra Graphs (CAG): Combining Context Dependency and Derivation Trees

In general, an expression that translates an input Sangam graph `G` to an output Sangam graph `G'` corresponds to an ordered set of context depen-

dency trees, derivation trees or a combination of context dependency and derivation trees. We use the term ***cross algebra tree*** (CAT) to refer to the individual trees and the term ***cross algebra graph (CAG)*** to refer to the graph representing this expression. Figure 12.8 gives an example of a cross algebra graph (*CAG*) that translates a given input Sangam graph G to an output Sangam graph G'. Here let e1:<A, B>, e2:<B, C>, e3:<C, D> and e4:<D, E> represent edges in the input Sangam graph G. Let e':<A', E'> represent an edge in the output Sangam graph G'. The cross algebra tree CAT in Figure 12.8 is given as:

$$\mathtt{CAT} = \mathtt{CAT3} \tag{12.10}$$

The output of the tree CAT3 is the final output of the CAT. The expression for CAT3 is given as follows:

$$\mathtt{CAT3} = \mathtt{DT1}, (\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op3}_{\mathtt{E'}}(\mathtt{E})) \tag{12.11}$$

The output of CAT3 is produced by the evaluation of its expression, namely, by the evaluation of the three inputs, namely the derivation tree DT1, and the two cross algebra operators op1 and op3. Using the order of evaluation for context dependency trees as specified in Section 12.2.2, the operators op1 and op3 must be evaluated prior to the evaluation of DT1. The evaluation of those two primitive operators op1 and op3 produces

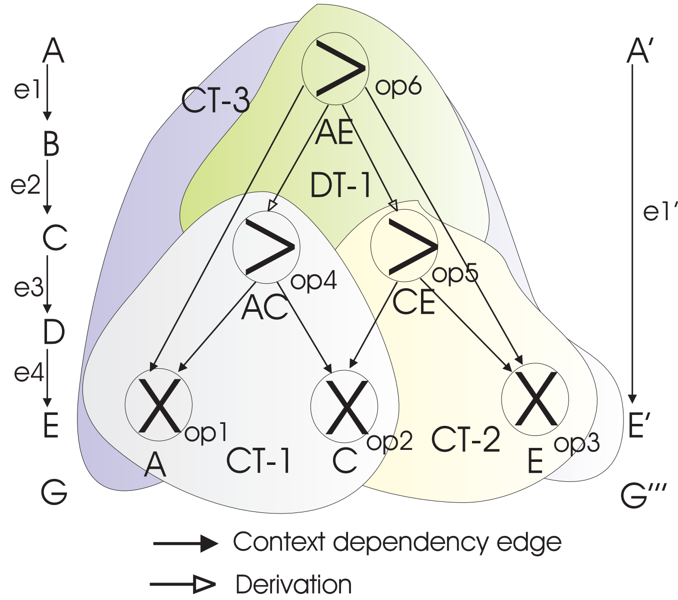


Figure 12.8: A Cross Algebra Tree.

the nodes A′ and E′ respectively. Given that op1 and op3 are in context dependency relation with the root of DT1, the outputs of op1 and op3 now become part of the final output of CAT3. The tree DT1 is given as:

$$\mathtt{DT1} = \mathtt{op6}_{\mathtt{e}':<\mathtt{A}',\mathtt{E}'>}(\mathtt{CT1}, \mathtt{CT2}) \tag{12.12}$$

The operator op6 represents the root of DT1, a derivation tree. The output of this derivation tree (and hence of op6) is the edge e′ between the nodes A′ and E′. This output is produced by the evaluation of the expression $\mathtt{op6}_{\mathtt{e}':<\mathtt{A}',\mathtt{E}'>}(\mathtt{CT1},\mathtt{CT2})$. Here the inputs for the operator op6 are the edges in the input graph produced by the context dependency trees CT1

and CT2 respectively. As per the definition of derivation (Definition 11), the temporary outputs (the output Sangam graph) generated by the context dependency trees CT1 and CT2 do not manifest themselves in the final output of the derivation tree. The expression for CT1 is given as follows:

$$\mathtt{CT1} = \mathtt{op4}_{\mathtt{eTemp1:<A',C'>}}(\mathtt{e1} :< \mathtt{A}, \mathtt{B} >, \mathtt{e2} :< \mathtt{B}, \mathtt{C} >), (\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op2}_{\mathtt{C'}}(\mathtt{C})) \tag{12.13}$$

Observation 1 *If there exists a derivation edge* $\twoheadrightarrow$ *from an operator* $\mathtt{op}_i$ *to the root operator* $\mathtt{op}_j$ *of a context dependency* $\mathtt{CT}_j$*, then the output Sangam graph* G' *produced by the context dependency tree* $\mathtt{CT}_j$ *serves as the input Sangam graph for the operator* $\mathtt{op}_i$*. The operator* $\mathtt{op}_i$ *directly consumes only one node or edge of Sangam graph* G' *(or two nodes and edges if* $\mathtt{op}_i = \ominus$*) and produces a node or an edge as output (or a node and two edges if* $\mathtt{op}_i = \ominus$*) (Definition 11).*

In the above example, op6 consumes the edge eTemp1 produced by the context dependency tree CT1 as well as the edge eTemp2, the output produced by CT2. The Sangam graph G' is an intermediate Sangam graph and is discarded, i.e, G' and the edge eTemp1:<A', C'> do not appear in the final output of op6. Similar to CT1, the expression for CT2 is given as:

$$\mathtt{CT2} = \mathtt{op5}_{\mathtt{eTemp2:<C',E'>}}(\mathtt{e3} :< \mathtt{C}, \mathtt{D} >, \mathtt{e4} :< \mathtt{D}, \mathtt{E} >), (\mathtt{op2}_{\mathtt{C'}}(\mathtt{C}) \circ \mathtt{op3}_{\mathtt{E'}}(\mathtt{E})) \tag{12.14}$$

To summarize the above example (Figure 12.8), the context dependency trees `CT1` and `CT2` produce two intermediate Sangam graphs `G'` and `G''` that smooth the edges `e1` and `e2` to produce edge `eTemp1` and smooth edges `e3` and `e4` to produce edge `eTemp2` respectively. The operator `op6` then gets its inputs from `CT1` and `CT2` and produces the edge `e':<A',E'>`. The operator `op6` also participates in the tree `CAT3`, the output of which is the final Sangam graph `G'''` with nodes `A'`, `E'` and the edge `e':<A', E'>`. Thus, the `CAT` in Figure 12.8 operates on the input Sangam graph `G` and produces as output the Sangam graph `G'''`.

As shown by the example above, operators may participate in more than one context dependency or derivation tree, i.e., they may appear multiple times in the CAT algebra expression. These *shared* operators are evaluated only once during the evaluation of the complete CAT expression, and hence contribute output once (versus multiple times) in the evaluation of the CAT. We say that two operators are the **same** if they observe the following conditions.

Definition 15 (Same Operators) *Two operators* $\mathtt{op}_i$ *and* $\mathtt{op}_j$ *are identified as being the same in one CAT, if they have the (1) same mapping type (cross, smooth, etc.); (2) same input node or edge (from the same input Sangam graph* `G`*) as identified by its label and identifier; and (3) produce as output the same node or edge in the* **same** *Sangam graph* `G'` *based on its label and identifier*[5]*.*

Definition 16 (Evaluation Semantics of Shared Operators) *If two operators* $\mathtt{op}_i$ *and* $\mathtt{op}_j$ *are the same as per Definition 15, then they are termed* shared *op-*

[5]Recall from Chapter 11 each node and edge in a Sangam graph has a unique identifier.

erators. Each shared operator is evaluated once, and contributes directly[6] *at most once to the final output graph* `G'` *produced by the CAT.*

Formally, we define a CAT expression as follows:

Definition 17 (CAT) *A CAT is an expression that operates on one or more input*

[6]Some other operator $\mathtt{op}_k$ may derive from it. The output of operator $\mathtt{op}_k$ may appear in the final output.

Sangam graphs `G` *and produces one or more output Sangam graph* `G'` *such that:*

$$\mathtt{CAT}_{\mathtt{o}(\mathtt{out}_\mathtt{o})}(\mathtt{in}_\mathtt{o}) = \begin{cases} DT_{i(out_i)}(in_i) & \text{where } DT_i \text{ is as per Definition 11} \\ CT_{i(out_i)}(in_i) & \text{where } CT_i \text{ is as per Definition 13} \\ (CAT_{j(out_j)}(in_j)) \quad , & \\ ((CAT_{k(out_k)}(in_k))) & \text{where } CAT_j \text{ is parent of } CAT_k \text{ such that } CAT_k \text{ must be evaluated prior to the evaluation of } CAT_j \text{ and } out_o = out_j \bigcup out_k^{\dagger}. \\ op_{j(out_j)} & \\ (CAT_{k(out_k)}(in_k)) & \text{where } op_j \text{ derives its input } in_j \text{ from the output } out_k \text{ produced by } CAT_k, \text{ and } out_o = out_j^{\ddagger}. \\ op_{j(out_j)}((CAT_{k(out_k)}(in_k)), & \\ (CAT_{l(out_l)}(in_l))) & \text{where } op_j \text{ derives its input } in_j \text{ from the outputs } out_k \text{ and } out_l \text{ produced by } CAT_k \text{ and } CAT_l \text{ respectively, and } out_o = out_j^{\gamma}. \end{cases}$$

$\dagger =$ A context dependency edge is added from the root $\mathtt{op}_j$ of $\mathtt{CAT}_j$ to the root $\mathtt{op}_k$ of $\mathtt{CAT}_k$.

$\ddagger =$ A derivation edge is added from $\mathtt{op}_j$ to $\mathtt{op}_k$, the root of $\mathtt{CAT}_k$.

$\gamma =$ A derivation edge is added from $\mathtt{op}_j$ to $\mathtt{op}_k$ and $\mathtt{op}_l$, the roots of $\mathtt{CAT}_k$ and $\mathtt{CAT}_l$ respectively..

Definition 18 *Given a CAT expression $\mathtt{CAT}_o$ as defined in Definition 17, a CAT tree $\mathtt{T}_{CAT} = (\mathtt{N}_o, \mathtt{E})$ such that (1) each node $\mathtt{n}_i \in \mathtt{N}_o$ represents a cross algebra operator ($\otimes$, $\ominus$, $\circleddash$, $\oslash$) in the expression $\mathtt{CAT}_o$; and (2) each edge $\mathtt{e}_i \in \mathtt{E}_d$ is either a derivation edge $\rightarrowtriangle$ or a context dependency edge $\rightarrow$ with semantics as defined in Definitions 12 and 14 respectively.*

Lemma 6 *A CAT tree $\mathtt{T}_{CAT}$ (Definition 18) given by the expression $\mathtt{CAT}_o$ (Definition 17) is a rooted, directed acyclic graph.*

Proof.

A CAT as defined in Definition 17 can be either a context dependency tree, a derivation tree or a composition of the two. We consider each case individually. Definition 15 defines the criteria for designating operators as shared operators. While at a practical level it is more efficient to physically share the operators across different trees, for the purpose of the discussion here, we assume that these shared (same) operators are physically distinct operators. That is there exist multiple copies of the same operator in the CAT[7]. In the forthcoming discussion we thus do not treat shared operators

[7] Note that in the CAT expression, we do already represent shared operators by multiple copies of the same operator.

as a special case.

- $\mathtt{CAT}_{out}\mathtt{(in)} = \mathtt{CT}_{i(out_i)}\mathtt{(in}_i\mathtt{)}$: We know by Lemma 5 that a context dependency tree is acyclic.

- $\mathtt{CAT}_{out}\mathtt{(in)} = \mathtt{DT}_{i(out_i)}\mathtt{(in}_i\mathtt{)}$: We know by Lemma 4 that a derivation tree is acyclic.

- $\mathtt{CAT}_{out}\mathtt{(in)} = \mathtt{CAT}_{j(out_j)}\mathtt{(in}_j\mathtt{)}, \mathtt{CAG}_{k(out_k)}\mathtt{(in}_k\mathtt{)}$: Let $\mathtt{CAG}_j$ and $\mathtt{CAG}_{k(out_k)}$ be any cross algebra trees such that they are acyclic. The expression $\mathtt{CAT}_{j(out_j)}\mathtt{(in}_j\mathtt{)}, \mathtt{CAG}_{k(out_k)}\mathtt{(in}_k\mathtt{)}$ denotes that there exists a context dependency edge from $\mathtt{op}_j$ the root of $\mathtt{CAT}_j$ to $\mathtt{op}_k$ the root of $\mathtt{CAG}_k$, making $\mathtt{CAG}_k$ a child of the root $\mathtt{op}_j$. As $\mathtt{CAG}_j$ is acyclic with all edges going from the root $\mathtt{op}_k$ to the children, and $\mathtt{CAT}_k$ is similar, it follows that after the addition of a context dependency edge from $\mathtt{op}_j$ to $\mathtt{op}_k$ in the same direction, $\mathtt{CAT}_{out}$ is also acyclic.

- $\mathtt{CAT}_{out}\mathtt{(in)} = \mathtt{op}_{j(out_j)}\mathtt{(CAT}_{k(out_k)}\mathtt{(in}_k\mathtt{))}$: Let $\mathtt{CAT}_k$ be a cross algebra tree that is acyclic. The expression $\mathtt{op}_{j(out_j)}\mathtt{(CAT}_{k(out_k)}\mathtt{(in}_k\mathtt{))}$ denotes that there exists a derivation edge from $\mathtt{op}_j$ to $\mathtt{op}_k$, where $\mathtt{op}_k$ is the root of $\mathtt{CAT}_k$. As $\mathtt{CAT}_k$ is acyclic with all edges going from the root $\mathtt{op}_k$ to the children, then it follows that after the addition of a derivation edge from $\mathtt{op}_j$ to $\mathtt{op}_k$, the edges are all still in the same direction. Hence, $\mathtt{CAT}_{out}$ is also acyclic.

- $\mathtt{CAT}_{out}\mathtt{(in)} = \mathtt{op}_{j(out_j)}\mathtt{((CAT}_{k(out_k)}\mathtt{(in}_k\mathtt{), (CAT}_{l(out_l)}\mathtt{(in}_l\mathtt{)))}$: Here the operator $\mathtt{op}_j$ is a binary operator with two inputs $\mathtt{CAT}_k$ and $\mathtt{CAT}_l$. The proof of this can given in a similar manner to the one above.

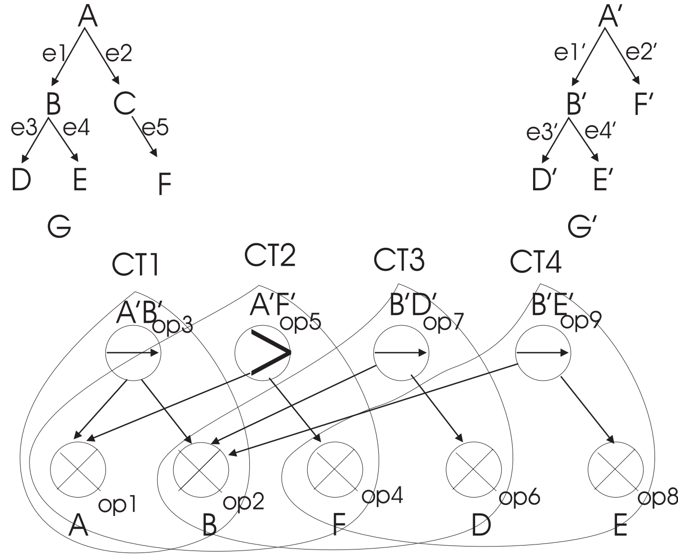


Figure 12.9: Another Example of a Cross Algebra Graph.

□

Consider now an example CAG in Figure 12.9. Here the `CAG` operates on input Sangam graph `G` and produces as output the Sangam graph `G'`. The edges of `G` are `e1`:<`A`, `B`>, `e2`:<`A`, `C`>, `e3`:<`B`, `D`>, `e4`:<`B`, `E`> and `e5`:<`C`, `F`>. Similarly for `G'`, the edges are `e1'`:<`A'`, `B'`>, `e2'`:<`A'`, `F'`>, `e3'`:<`B'`,`D'`>, and `e4'`:<`B'`,`E'`>. The expression for this is given by the following CAG:

$$\texttt{CAG} = (\texttt{CT1} \circ \texttt{CT2} \circ \texttt{CT3} \circ \texttt{CT4}) \tag{12.15}$$

where $\texttt{CT1} = \texttt{op3}_{(e1')}(\texttt{e1}), (\texttt{op1}_{\texttt{A}'}(\texttt{A}) \circ \texttt{op2}_{\texttt{B}'}(\texttt{B}))$

$\texttt{CT2} = \texttt{op5}_{(e2')}(\texttt{e2}, \texttt{e5}), (\texttt{op1}_{\texttt{A}'}(\texttt{A}) \circ \texttt{op4}_{\texttt{F}'}(\texttt{F}))$

$\texttt{CT3} = \texttt{op7}_{(e3')}(\texttt{e3}), (\texttt{op2}_{\texttt{B}'}(\texttt{B}) \circ \texttt{op6}_{\texttt{D}'}(\texttt{D}))$ and

$\texttt{CT4} = \texttt{op9}_{(e4')}(\texttt{e4}), (\texttt{op2}_{\texttt{B}'}(\texttt{B}) \circ \texttt{op8}_{\texttt{E}'}(\texttt{E}))$.

Here note that the operators op2 and op1 are *shared* operators and hence are used multiple times in individual context trees. However, as per the evaluation semantics of shared operators (Definition 16), these operators are only evaluated once during the evaluation of the CAG, and their output only appears once in the output Sangam graph G'. Also note that the output of context dependency trees CT1, CT2, CT3 and CT4 all appear in the final output of the CAG.

Based on the definition of a CAT (Definition 17), we now define a cross algebra graph (CAG).

Definition 19 (CAG) *A cross algebra graph (CAG) is an expression that operates on one or more input Sangam graphs* G *and produces one or more output Sangam graph* G' *such that:*

$$\texttt{CAG}_{\texttt{o}(\texttt{out}_\texttt{o})}(\texttt{in}_\texttt{o}) = \begin{cases} (CAT_{j(out_j)}(in_j))\ [\circ & \text{where } CAT_j \text{ and } CAT_k \text{ are sib-} \\ (CAT_{k(out_k)}(in_k))]^+ & \text{ling CATs such that } out_o = out_j \\ & \bigcup out_k \end{cases}$$

Definition 20 *Given a CAG expression* $\texttt{CAG}_o$ *as defined in Definition 19, a CAG is a forest* $\texttt{T}_{CAG}$ *of trees* $\texttt{CAT}_i$ *where each tree* $\texttt{CAT}_i$ *is as given in Definition 18.*

Lemma 7 *A CAG as defined in Definition 20 is a directed acyclic graph.*

Proof. We know by Lemma 6 that a CAT is acyclic. The symbol $\circ$ denotes the sibling relation, i.e., no new edge exists between the two CATs $\mathtt{CAT}_j$ and $\mathtt{CAT}_k$ to model this sibling relationship. Hence, $\mathtt{CAG}_{out}$ is cycle free. □

CAG Local Correctness. Most algebras, relational or XML for example, are guaranteed to produce valid relational tables or valid XML fragments respectively. For example, all relational algebra operators operate on one or two input relations and are guaranteed to produce a well-formed relation as output. The case for XML algebra is similar. We now define a *well-formed* expression for a CAG.

Definition 21 (Well-Formed CAG Expression) *We say that a CAG expression by Definition 19 is* **well-formed** *if it produces as output a Sangam graph* $\mathtt{G'}$ *such that:*

1. *for all nodes* $\mathtt{n}_i' \in \mathtt{G'}$, $\tau(\mathtt{n}_i') \in \mathcal{N}$*;*
2. *for all edges* $\mathtt{e}_i' \in \mathtt{G'}$, $\tau(\mathtt{e}_i') \in \mathcal{E}$*;*
3. *for all edges* $\mathtt{e}_i'$ *:*$<\mathtt{n}_i', \mathtt{n}_j'> \in \mathtt{G'}$, $\mathtt{n}_i' \in \mathtt{G'}$ *and* $\mathtt{n}_j' \in \mathtt{G'}$.

Given a Sangam graph $\mathtt{G}$ (Definition 6), we know by definition of the cross algebra operators (Definitions 7, 8, 9 and 10), that each algebra operator is guaranteed to produce nodes or edges in the output Sangam-graph $\mathtt{G'}$ such that their types are the same as the nodes and the edges from which they are mapped. For example, if an input node has a type $\tau(\mathtt{n}) \in \mathcal{N}$, then the output node $\mathtt{n'}$ produced by mapping $\mathtt{n} \in \mathtt{G}$ via a cross algebra operator also has $\tau(\mathtt{n'}) \in \mathcal{N}$. Hence, by the definitions of cross algebra

operators we know that a CAG always guarantees conditions (1) and (2) of well-formedness of CAGs.

By definition of the connect, smooth and subdivide operators (Definitions 8, 9 and 10), we know that an edge $\mathtt{e{:}\langle m,\ n\rangle} \in \mathtt{G}$ can be mapped via a cross algebra operator $\mathtt{op} \in \{\ominus, \varolessthan, \varogreaterthan\}$ to an edge $\mathtt{e'{:}\langle m',\ n'\rangle} \in \mathtt{G'}$ if and only if the two end-points of the edge $\mathtt{e}$, $\mathtt{m}$ and $\mathtt{n}$ with $\mathtt{m}, \mathtt{n} \in \mathtt{G}$, are also mapped over and already exist in $\mathtt{G'}$. That is, $\mathtt{m'} = \otimes(\mathtt{m})$ and $\mathtt{n'} = \otimes(\mathtt{n})$, and $\mathtt{m'}, \mathtt{n'} \in \mathtt{G'}$. Condition (3) of the well-formedness of the CAG can be checked statically by examining the output of each operator in the CAG.

Thus, given a valid Sangam graph $\mathtt{G}$, a well-formed CAG expression as per Definition 21 produces a new Sangam graph $\mathtt{G'}$ as output, that can be shown to be structurally valid algorithmically.

CAG Application Correctness. A purpose of the CAGs and the Sangam graphs is the ability to now enable the mapping of relational schemas to XML and vice versa in a principled fashion. In Chapter 11 we have presented algorithms to translate a relational schema and data to a Sangam graph and vice versa. We have also given similar algorithms for the XML DTD and documents. These algorithms however impose a restriction on the Sangam graphs , i.e., they require the input Sangam graphs to be connected. It is therefore an important criterion for the CAG to generate not only valid Sangam graphs but to also ensure that they are connected. We term this as the *output connectivity* of a CAG expression.

Definition 22 *The CAG expression is said to be* output-connectivity compli-

ant, *if given a valid and connected Sangam graph* G *it produces a valid and connected output Sangam graph* G' *such that: for any node* $n_i' \in G'$ *there is a path* p_i':<r', n_i'> *with* r' *one of the root nodes of the Sangam graph with the path* p_i' *a sequence of directly connected edges from* r' *to* n_i'. *That is the Sangam graph* G' *is connected.*

In general we cannot guarantee that a given CAG expression is/or will be output-connectivity compliant. However, this output-connectivity can be determined by examining the output Sangam graph G' produced by the CAG. Many algorithms have been proposed in graph theory literature that calculate the connectivity of a graph. A common set of algorithms for this are the *max-flow* based algorithms. The max-flow based vertex connectivity algorithm as given in [GY98] calculates the node connectivity κ(G) of a graph G= (N, E). This algorithm can be employed to discover if the output Sangam graph G' is a connected graph.

Thus, given a valid and connected input Sangam graph G, a well-formed and output-connectivity compliant CAG expression can be shown to produce as output a valid and connected Sangam graph G' algorithmically. The algorithms presented in Chapter 11 can now be applied to the valid and connected output Sangam graph G' to produce the mapped application schema and data.

12.3 Evaluating the Cross Algebra Operators - The Physical Plan

The cross algebra operators are a set of algebra operators that enable the mapping of a schema in one data model to a schema in another data model. In practice, to facilitate the execution of these logical operators we need to consider (1) the binding of these operators to physical algebra operators which includes the binding of these operators to an input and output data model; and (2) given the physical operators, the general algorithm for executing the algebra expression. We use a one-pass algorithm to execute the algebra operators, i.e., we read data segments into main memory and transform them, and then write out the data segments. In this section we present the physical operators, and the evaluation algorithm for a CAG.

12.3.1 Physical Algebra Operators

For each cross algebra operator, we define a physical equivalent of the operator. In addition, we also define additional physical operators for reading and constructing the objects from the input data (`scan`), iterating over the input (`open` , `getNext` , `close`), copying (`copy`) and writing (`write`) the output data.

Iterators. The iterator operators are identical to the iterators provided for the relational algebra [UW97]. The `open` operator gets a handle on the extent of a node; the `getNext` operator iterates over the extent of the node and returns one object at a time (called the *current* object). The `close` op-

erator closes the handle on the node's extent; the `copy` operator makes a copy of the specified object; and `write` (`o`, I (`n`)) writes the specified object `o` to the output extent I (`n`). The `write` operator writes one object at a time.

Scan. Typically in relational systems, there are two basic approaches to locating the tuples of a relation `R` - ***table scan*** and ***index scan*** [UW97]. Both of these operations bring into main memory, the tuples of the relation `R` based on either its block storage or an index on its attribute. For a given Sangam graph, , input objects (data) can be brought into memory based on the storage of the input node, i.e., bring into memory the extent of the node.

The operator ***scan*** operates on an input node `n` or an input edge `e`. It scans the extent of node `n` (or the edge `e`) and brings it into main memory.

Cross Algebra Operators. Using the scan and iterator operators to read and iterate over the input objects, we now show how the cross algebra operators presented in Section 12.1 can be implemented. Figures 12.10, 12.11, 12.12 and 12.13 depict the algorithms in pseudo-code for the cross algebra operators: cross, connect, smooth, and subdivide respectively. Here as in Chapter 11, we assume that we have functions such as insertNode that creates a Sangam graph node, insertEdgeAt [8] that creates an edge between two given nodes, and get and set functions that allow us to retrieve the properties of a node and an edge in the given Sangam graph. Tables 12.2, 12.3 and 12.4 summarize the functions used in these algorithms.

Here we briefly describe the pseudo-code for the smooth operator. Fig-

[8] Full description of insertNode and insertEdgeAt operations can be found in Chapter 14.

Function	Return Type	Description
n.getLabel()	String	Returns the label of the node n
I (n)	Set<Object>	Returns the extent of the node n
G′.nodeMapping(n)	Node	Returns the node n′ ∈ G′ such that node n is mapped to node n′
e.getLabel()	String	Returns the label of the edge e
e.getQuantifier()	Quantifier	Returns the quantifier annotation of the edge e
e.getOrder()	Order	Returns the order annotation of the edge e
R (e)	Set<Object>	Returns the extent of the edge e
e.fromNode()	Node	Returns the node n from which the edge e stems
e.toNode()	Node	Returns the node n on which the edge e is incident
G′.edgeMapping(e)	Edge	Returns the edge e′ ∈ G′ such that edge e is mapped to edge e′

Table 12.2: List of Methods Used in the Pseudo-Code for the Algebra Operators.

ure 9 graphically depicts the functionality of the smooth operator. Here, node n_1 is already mapped to $n_1{}'$, node n_2 is not mapped over, and node n_3 is mapped to node $n_3{}'$ in output Sangam graph G′. Consider the pseudo-code for the smooth operator as shown in Figure 12.12. The goal of this algorithm and hence the smooth operator is to create an edge e′ between the nodes $n_1{}'$ and $n_3{}'$ in the output Sangam graph G′ such that all objects $o_x \in I(n_1)$ that are in relationship with objects $o_z \in I(n_3)$ via some object $o_y \in I(n_2)$ are translated such that their exists a direct relationship between

Function	Return Type	Description
o_n.*getValue*()	Value	Returns the value of the node object
o_e.*fromObject*()	NodeObject	Returns the NodeObject from which the edgeObject o_e stems.
o_e.*toObject*()	NodeObject	Returns the NodeObject on which the edgeObject o_e is incident.
o_t.*getParentObject*()	NodeObject	Returns the NodeObject for which this object-tuple is defined. For example if $o_t = o$:<o_1, o_2, ..., o_n>, then *getParentObject*() returns o.
o_t.*getObject*()	NodeObject	Returns the NodeObject at index position indexPos in the ObjectTuple. The order of each object denotes the order of the edges in which they participate.

Table 12.3: List of Methods Used in the Pseudo-Code for the Algebra Operators.

corresponding objects o_x' and o_z'.

Let e_1:<n_1, n_2> and e_2:<n_2, n_3> be two edges between the nodes n_1, n_2 and $n_3 \in G$. The smooth operator creates a new edge $e' \in G'$ between the nodes n_1' and n_3'. For every edgeObject e_{o1}:<o_1, o_2> ∈ R (e_1), we find all the edgeObjects e_{o2}:<o_i, o_j> ∈ R (e_2), such that $o_i = o_2$. The smooth creates a new edgeObject o_e:<o_1',o_j'> for every e_{o2}. The edgeObject is inserted into the extent R (e') of the new edge e'.

Function	Return Type	Description
op.hasChildren()	Boolean	Returns true of the operator op has children operators. A child operator is either related via a context dependency relation or a derivation relation.
op.getNextChild()	Operator	Returns the next child operator of the current operator op
op.evaluate()	Sangam graph	Evaluates the operator based on the set of inputs provided. Returns the output Sangam graph. .
op.markDone()	none	Marks that the operator has been evaluated.
op.edgeType()	String	Returns the type of edge between the operator op and the passed in operator. Valid answers are *derivation* or *contextDependency*.

Table 12.4: List of Methods Used in the Algorithms for Evaluating Cross Algebra Graphs.

12.3.2 Evaluating a Cross Algebra Graph

As presented in Section 12.2, the cross algebra graph consists of either context dependency or derivation trees or a composition of context dependency and derivation trees. Thus CAGs can be composed as a forest of context dependency and derivation trees, a larger derivation or a larger context dependency tree. Each tree $\mathtt{CAT}_i$ of the CAG, called a CAT, can be evaluated independently of any other tree $\mathtt{CAT}_j$ of the CAG. To evaluate

```
operator crossOp (input: Node n)
{
    Node n' ← insertNode ( n.getLabel(), τ)
    Extent I(n) ← scan (n)

    NodeObject o_x
    I(n) .open ()
    while (o_x ← I(n) .getNext ()) {
      NodeObject o_x' ← copy (o_x)
      write (o_x', I (n'))
    }
    I(n) .close ()
}
```

Figure 12.10: The Cross Physical Operator - An Implementation.

a CAT, we use post-order evaluation, i.e., all children operators $\texttt{op}_j$ of an operator $\texttt{op}_k$ are evaluated prior to the evaluation of $\texttt{op}_k$.

If the edge between the child operator $\texttt{op}_j$ and the parent operator $\texttt{op}_i$ is a context dependency edge (denoted by the symbol "," in the CAG expression), then the final output produced by the parent is a union of its output and the output of all its context dependency children. Thus, in this case $\texttt{out}_{final} = \texttt{out}_i \bigcup \texttt{out}_j$, where $\texttt{out}_{final}$ is the final output returned by the parent operator $\texttt{op}_i$, $\texttt{out}_i$ the local output of $\texttt{op}_i$ and $\texttt{out}_j$ is the output returned by the child operator $\texttt{op}_j$. To facilitate shared operators, each operator $\texttt{op}_j$ is marked "visited" the first time it is evaluated, and its local output is cached. If the operator $\texttt{op}_j$ is re-visited, no further evaluation of $\texttt{op}_j$ is done, and its cached output is returned to the invoking parent operator $\texttt{op}_i$.

If the edge between the child operator $\texttt{op}_j$ and the parent operator $\texttt{op}_i$ is a derivation edge (denoted by the symbol "( )" in the CAG expression),

```
operator connectOp (input: Sangam graph G, input Edge e:<n1, n2>,
        output: Sangam graph G')
{
    Node n1' ← G'.nodeMapping(e.fromNode)
    Node n2' ← G'.nodeMapping(e.toNode)

    // Create a new edge and insert it into G'
    Edge e' ← createSAGEdge( e.getLabel(), e.getQuantifier(), e.getOrder(),
            n1', n2')
    G'.insertEdge (n1', n2', e')

    Extent Ie ← scan (e)

    EdgeObject oe
    Ie.open ()
    while (oe← Ie.getNext ()) {
      EdgeObject oe' ← new EdgeObject ()
      oe'.fromObject ← G'.objectMapping(oe.fromObject)
      oe'.toObject ← G'.objectMapping(oe.toObject)
      write (oe', R (e'))
    }
    Ie.close ()
}
```

Figure 12.11: The Connect Physical Operator - An Implementation.

then the output $\texttt{out}_j$ of the child operator $\texttt{op}_j$ is consumed directly by the parent operator $\texttt{op}_i$, i.e., output $\texttt{out}_j$ of the operator $\texttt{op}_j$ is the input $\texttt{in}_i$ of the operator $\texttt{op}_i$. The final output $\hat{\texttt{out}}_{final}$ produced by $\texttt{op}_i$ in this case is its own local output. Thus, in this case $\texttt{out}_{final} = \texttt{out}_i$, where $\texttt{out}_{final}$ is the final output returned by the parent operator $\texttt{op}_i$ and $\texttt{out}_i$ the local output of $\texttt{op}_i$.

In both of the above cases (derivation and context dependency), in order to facilitate shared operators, each operator $\texttt{op}_j$ is marked "visited" the first time it is evaluated, and its local output is cached. If the operator

```
operator smoothOp (input: Sangam graph G, input: Edge e1:<n1, n2>,
      input: Edge e2:<n2, n3>, output: Sangam graph G')
{

    Node n1' ← G'.nodeMaping(e1.fromNode)
    Node n3' ← G'.nodeMapping(e2.toNode)

    // Create a new edge and insert into G'
    Edge e' ← createSAGEdge( e1.getLabel(), e1.getQuantifier() *
          e2.getQuantifier(), e1.getOrder(), n1', n3')

    G'.insertEdge (n1', n3', e')

    Extent Ie1 ← scan (e1)
    Extent Ie2 ← scan (e2)

    EdgeObject oe1
    EdgeObject oe2

    Ie1.open ()
    Ie2.open ()

    while (oe1 ← Ie1.getNext ()) {
      foreach (oe2 ← Ie2.getNext ()) {
        if (oe1.toObject = oe2.fromObject) {
          EdgeObject oe' ← new EdgeObject ()
          oe'.fromObject ← G'.objectMapping(oe1.fromObject)
          oe'.toObject ← G'.objectMapping(oe2.toObject)
          write (oe', R (e'))
        }
      }
    }
    Ie1.close ()
    Ie2.close ()
}
```

Figure 12.12: The Smooth Physical Operator - An Implementation.

op_j is re-visited, no further evaluation of op_j is done, and its cached output is returned to the invoking parent operator op_i. Evaluation of the tree terminates with the evaluation of the root operator. Figure 12.14 gives the

```
operator subdivideOp (input: Edge e1:<n1, n3>, output: Sangam graph G')
{
    Node n1' ← G'.nodeMapping(e1.fromNode)
    Node n3' ← G'.nodeMapping(e1.toNode)

    // Create the required Node and Edges and insert in G'
    Node n2' ← createSAGNode( defaultLabel)
    Edge e1' ← createSAGEdge( e1.getLabel(), e1.getQuantifier(), e1.getOrder(),
          n1', n2')
    G'.insertNode (n2')
    G'.insertEdge (n1', n2', e1')

    Edge e2' ← createSAGEdge( e1.getLabel(), 0:1, 1, n2', n3')

    G'.insertEdge (n2', n3', e2')

    Extent Re1 ← scan (e1)

    Re1.open ()
    EdgeObject oe

    while (oe ← Re1.getNext ()) {

      NodeObject ox' ← new NodeObject ()

      EdgeObject oe' ← new EdgeObject ()
      oe'.fromObject ← G'.nodeMapping(oe.fromObject)
      oe'.toObject ← ox'
      write (oe', R (e1'))

      EdgeObject of' ← new EdgeObject ()
      of'.fromObject ← ox'
      of'.toObject ← G'.nodeMapping(oe.toObject)
      write (of', R (e2'))

    }
    Re1.close ()
}
```

Figure 12.13: The SubDivide Operator - An Implementation.

algorithm for evaluating the cross algebra graph.

```
function EvaluateCAG (input: CAG cag, input: Sangam graph G,
          output: Sangam graph G')
{
    List roots ← cag.getRoots()
    while (roots != null) {
      operator op ← roots.getNext()
      EvaluateCAT (op, G, G')
    }
}

function EvaluateCAT (input: Operator op, input: Sangam graph G,
          output: Sangam graph G')
{
    if (!op.hasChildren())
      Sangam graph G' ← op.evaluate(G, G')
      op.markDone()
      Sangam graph out ← G' // cache the local output
      return localG'

    while (op.hasChildren()) {
      operator opC ← op.getNextChild()
      if (e : <op, opC> = derivation)
          SAG G_local ← EvaluateCAT (opC, G, G')
          SAG G' ← op.evaluate(G_local, G')
          op.markDone()
          SAG out ← G' // local cached output
          return G'

      elseif (e : <op, opC> = context dependency)
          SAG G_local ← EvaluateCAT (opC, G, G')
          SAG G'_local ← op.evaluate(G, G')
          SAG G' ← G_local ⋃ G'_local
          op.markDone()
          SAG out ← G'_local // local cached output
          return G'
    }
}
```

Figure 12.14: The Evaluation Algorithm for a Cross Algebra Graph.

Evaluation Termination.

Lemma 8 (CAG Termination) *The evaluation of a CAG as defined by Definition 19 will always terminate.*

Proof: Let CAG represent a CAG as defined in Definition 19. For termination, we consider the following cases:

- Case 1 - CAG = CT: A context dependency tree CT expression is evaluated from the right to left, that is from the leaf operators (children operators) to the parent operator. As each context dependency tree CT has at most one root and CT is free of cycles (Lemma 5), the evaluation of CT terminates when the root operator is evaluated.
- Case 2 - CAG = DT: A derivation tree DT expression is evaluated from the inner most operator, represented as leaves of the tree, to the outermost operator, the root of the tree. Evaluation terminates when the root operator is evaluated. Again, as a derivation tree contains no cycles (Lemma 4), the evaluation is always guaranteed to terminate.
- Case 3 - CAG = op_i, (CAG_j) : op_i,(CAG_j) represents a context dependency tree where op_i is the root of the CAG CAG_j, such that there exists a context dependency edge from the root operator op_i to the root op_j of the CAG CAG_j.
    1. CAG_j = CT: In this case CAG represents a large but pure context dependency tree such that the operator op_i is the root of the tree. This thus reduces to Case 1. Hence the evaluation of CAG will always terminate in this case.

2. $\texttt{CAG}_j = \texttt{DT}$: We know by Case 2 that the evaluation of a derivation tree always terminates at the root of the derivation tree $\texttt{op}_j$. As the output of $\texttt{op}_j$ and $\texttt{op}_i$ together present a context dependency tree the evaluation of CAG will terminate with the evaluation of $\texttt{op}_i$ (Case 1). Hence the evaluation of CAG will terminate in this case.

- Case 4 - $\texttt{CAG} = \texttt{op}_i\texttt{(CAG}_j\texttt{))}$: $\texttt{op}_i(\texttt{CAG}_j)$ represents a derivation tree where $\texttt{op}_i$ is the root of the CAG $\texttt{CAG}_j$, such that there exists a derivation edge from the root operator $\texttt{op}_i$ to the root operator $\texttt{op}_j$ of $\texttt{CAG}_j$.

  1. $\texttt{CAG}_j = \texttt{DT}$: In this case CAG represents a large but pure derivation tree such that the operator $\texttt{op}_i$ is the root of the tree. This thus reduces to Case 2. Hence the evaluation of CAG will always terminate in this case.

  2. $\texttt{CAG}_j = \texttt{CT}$: We know by Case 1 that the evaluation of a context dependency tree always terminates at the root of the context dependency tree $\texttt{op}_j$. As the output of $\texttt{op}_j$ is the input of $\texttt{op}_i$, the evaluation of CAG will terminate with the evaluation of $\texttt{op}_i$. Hence the evaluation of CAG will always terminate in this case.

- Case 5 - $\texttt{CAG} = \texttt{CAG}_i \circ \texttt{CAG}_j$: We know from the above cases that $\texttt{CAG}_i$ and $\texttt{CAG}_j$ always terminate. Moreover, we know from Lemma 7 that CAG is acyclic and there exists no edges between $\texttt{CAG}_i$ and $\texttt{CAG}_j$. The evaluation of CAG terminates with the evaluation of $\texttt{CAG}_i$ and $\texttt{CAG}_j$. Hence evaluation of CAG will terminate. □

12.4 An Example

Shanmugasundaram et al. [STZ+99] have proposed the *basic inlining* technique for translating an XML DTD and associated document into a relational schema and data. In this section, we describe the basic inlining technique and then show how it can be represented by a cross algebra graph.

12.4.1 The Basic Inlining

The Basic Inlining Technique. For each element, the basic inlining technique inlines as many descendants of an element as possible into a single relation. However, as an XML document can be rooted at any element, this technique is applied for all elements in a DTD and hence creates relations for every element. As an example consider the XMark benchmark DTD given in Figure 10.2. Here the `item` element would be mapped to a relation with attributes `location`, `mailbox` and `name`. Set-valued attributes such as the attribute `mail` in element `mailbox` are stored following the standard technique for storing sets in a RDBMS. Hence, a relation is created for the `mail` attribute and it is linked to the `mailbox` using a foreign key. Figure 12.15 shows the relational schema obtained by applying the basic inlining technique to the fragment of XMark benchmark DTD in Figure 10.2.

Shanmugasundaram et al. [STZ+99] present an algorithm that first converts a given DTD into an element graph. This element graph is constructed as follows. Do a depth first traversal of the DTD graph, starting at the element node for which the relations are being constructed. Each node

is marked as "visited" the first time it is reached and is unmarked once all its children have been traversed.

If an unmarked node in the DTD graph is reached during the depth first traversal, a new node bearing the same name is created in the element graph. In addition, a *regular* edge is created from the most recently created node in the element graph with the same name as the DFS parent of the current DTD node to the newly created node.

If an attempt is made to traverse an already marked DTD node, then a *backpointer* edge is added from the most recently created node in the DTD graph to the most recently created node in the element graph with the same name as the marked DTD node.

In general, given an element graph, relations are created as follows. A relation is created for the root element of the graph. All the element's descendants are inlined into that relation with the following two exceptions: (a) children with a "*" are made into separate relations - this corresponds to creating a new relation for a set-valued child; and (b) to handle recursion within elements, each node in the element graph having a backpointer edge is made into a separate relation. For further details we refer the reader to [STZ$^+$99].

Cross Algebra Graph for Basic Inlining. To construct the basic inlining technique as a cross algebra graph (CAG) we follow a simple algorithm. Given an input Sangam graph that represents the element graph constructed using the algorithm above [STZ$^+$99], we create:

1. cross ($\otimes$) operators for every node in the Sangam graph. This includes

ITEM(id VARCHAR(10) NOTNULL, featured VARCHAR(100), item.location VARCHAR(200), item.mailbox.id VARCHAR(10), item.name.lastName VARCHAR(200), item.name.firstName VARCHAR(200))

ITEM.MAILBOX.MAIL(text VARCHAR(800), item.mailbox.mail.from VARCHAR(50), item.mailbox.mail.to VARCHAR(50), item.mailbox.mail.date VARCHAR(8), parentId INTEGER)

LOCATION(location VARCHAR(200), parentId INTEGER)

MAILBOX(id VARCHAR(10), parentId INTEGER)

MAIL(text VARCHAR(800), from VARCHAR(50), to VARCHAR(50), date VARCHAR(8), parentId INTEGER)

FROM(from VARCHAR(200), parentId INTEGER)

TO(to VARCHAR(200), parentId INTEGER)

DATE(date VARCHAR(200), parentId INTEGER)

NAME(firstName VARCHAR(200), lastName VARCHAR(200), parentId INTEGER)

FIRSTNAME(firstName VARCHAR(200), parentId INTEGER)

LASTNAME(lastName VARCHAR(200), parentId INTEGER)

Figure 12.15: Fragment of Relational Schema Generated by Mapping the XMark Benchmark Schema of Figure 10.2 using the Basic Inlining Technique [STZ$^+$99]. All field sizes are set during the mapping of atomic nodes which are not shown here.

nodes that represent elements as well as attributes. Thus, considering the Sangam graph in Figure 11.3 that represents the XMark DTD of Figure 10.2, $\otimes$ operators will be created for the nodes labeled `item`, `location`, `featured`, `id`, etc.

2. The mapping of the element content, i.e., its (sub) elements and its

attributes, is represented by a collection of connect, smooth and subdivide nodes. Here, the path p from a given node m to another node n is given by a sequence of edges $p = e_1.e_2.\ldots.e_n$. The size of the path p, $|p| = n$. We ignore the backpointer edges unless explicitly specified. In the steps below, we calculate the path from given node m to a leaf node. Recall that a leaf node is any node n that only has atomic children (11). For clarity of the algorithm given below, we assume here that a leaf is also any node $\hat{n}$ such that the quantifier annotation on the edge e:<m, n> is either [0:i] or [1:i], $i \neq 0$ or 1 [9], and the node m is the parent node. For each path $p = e_1.e_2.\ldots.e_n$ we do:

- if $p = e_1$, and $|p| \leq 1$ then, create a connect ($\ominus$) operator that maps the edge e_1:<m, n> to an edge e_1' in the output Sangam graph G'. Context dependency edges are built from the connect operator to the cross operator that maps the node m as well as to the cross operator that maps the node n. Consider the Sangam graph in Figure 12.16. Here a $\ominus$ operator is created for the edge between the nodes labeled item and location. Also the $\ominus$ operator will have a context dependency edge to the $\otimes$ operators that map these nodes item and location. Figure 12.17 depicts the context dependency tree produced by the above step for the input Sangam graph G in Figure 12.16, while Figure 12.18 shows the output Sangam graph G'.

[9] In XML terms this would be any element with a + or *.

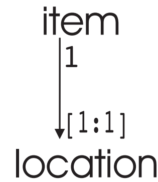


Figure 12.16: Fragment of the Input Sangam graph G.

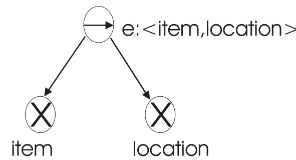


Figure 12.17: The Context Dependency Tree Produced by the Basic Inlining Technique for the Input Sangam graph in Figure 12.16.

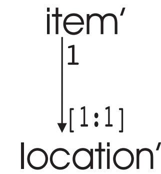


Figure 12.18: The Resultant Fragment of the Output Sangam-graph G' Produced by the Evaluation of the CAT in Figure 12.17 Defined on the Input Sangam graph in Figure 12.16.

- if $p = e1.e2$, and $|p| = 2$, then create smooth ⊝ operator that takes as input the two edges e_1 and e_2, and produces one edge e' in the output Sangam graph G'. Context dependency edges are built from the ⊝ operator to the ⊗ operators that map the end-points (nodes) of the path. Consider the input Sangam-graph G in Figure 12.19. Here the path from the node `item` to the node `firstName` via the node `name` is of size 2. This is due to the fact that there is one edge e_1 between the nodes `item` and `name` and another edge e_2 from the node `name` to the node `firstName`. A ⊝ operator is built to map the edges e_1 and e_2 to a new edge e_1'. Context dependency edges are built from the ⊝ operator to the ⊗ operators that map the nodes `item` and `firstName` (the end-points of the path). Figure 12.20 represents the cross algebra tree for input Sangam graph G in Figure 12.19, and Figure 12.21 depicts the output Sangam graph G'.

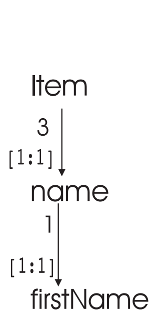


Figure 12.19: Fragment of the Input Sangam graph G.

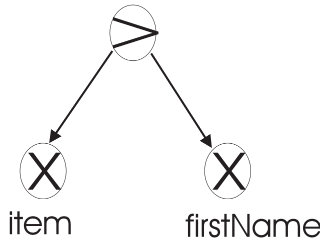


Figure 12.20: The Context Dependency Tree Produced by the Basic Inlining Technique for the Input Sangam graph in Figure 12.19.

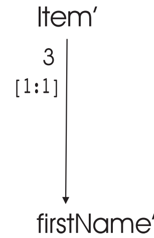


Figure 12.21: The Resultant Fragment of the the Output Sangam graph G′ Produced by the Evaluation of the CAT in Figure 12.20.

- if $\mathtt{p} = \mathtt{e}_1.\mathtt{e}_2.\ldots.\mathtt{e}_n$, and $|\mathtt{p}| > 2$, then to inline the edges and produce one edge $\mathtt{e'}{:}{<}\mathtt{m'},\mathtt{n'}{>}$ where m′ and n′ are the mapped end-points of the path p, we build derivation trees of smooth ⊗ and connect ⊖ operators in the manner detailed below.
  - The input path p is broken down into paths $\mathtt{p}_1$, $\mathtt{p}_2 \ldots \mathtt{p}_m$ of size 2. A ⊗ operator is created for every such pair of edges. For paths of odd size, a connect ⊖ operator is created for the path $\mathtt{p}_m$ with $|\mathtt{p}_m| = 1$. Thus for example, if the path size $|\mathtt{p}| = 4$, two ⊗ operators will be created, where as if $|\mathtt{p}| = 5$, then two ⊗ operators and one ⊖ operator is created. Context dependency edges between the operators are built as described in step 3. An intermediate output Sangam graph G′ will be created here, such that the size of the path p′ in the Sangam graph G′, is $|\mathtt{p'}| = \mathtt{n}/2$ if $|\mathtt{p}|$ was even and $|\mathtt{p'}| = \mathtt{n}/2 + 1$ if $|\mathtt{p}|$ was odd.

– Next, to further inline the path p′ and to produce a path $p^{n/4}$ of size n/4, one ⧁ operator op_s is created for every pair of ⧁ and ⊖ operators op_1, op_2 created in the previous step. A derivation edge is built from the operators op_1 and op_2 to the operator op_s to indicate that the operator's inputs are the outputs of the two operators op_1 and op_2.

– The above step is repeated until the an output Sangam graph G′ is produced with a path p^n of size 1.

– As a last step, we create the context dependency edges from the root of the CAT (this is operator with the final output of one edge) to the two end-points of the edge as described in step 3.

Consider the input Sangam graph fragment G shown in Figure 12.22. To inline the leaf node firstName, we create a smooth ⧁ operator op_1 that maps the input edges e_1:<item, name>, and e_2:<name, first>. This produces the intermediate edge e′<item, first>. A connect ⊖ operator op_2 maps over the edge e_3:<first, firstName>. The output of this operator, edge e′′:<first, firstName> and the output of op_1 become input edges for smooth (⧁) operator op_3 that produces edge e′′′:<item, first>. The expression for the cross algebra graph CAG in Figure 12.23 is given as: CAG = ($op6_{e'}$ (CT1, CT2)), ($op1_{item'}$(item) ∘ $op3_{firstName'}$(firstName))). the context dependency tree CT1 is given as CT1 = ($op4_{e1Temp}$(e1: <item,

`name`>), ($\texttt{op1}_{\texttt{item}'}$(`item`) $\circ$ $\texttt{op2}_{\texttt{name}'}$(`name`)), and context dependency tree `CT2` is given as `CT2` = ($\texttt{op5}_{\texttt{e2Temp}}$ (`e2`: <`name`, `firstName`>), ($\texttt{op2}_{\texttt{name}'}$ (`name`) $\circ$ $\texttt{op3}_{\texttt{firstName}'}$(`firstName`)).

Figure 12.23 depicts the cross algebra tree produced for the input Sangam graph in Figure 12.22 and Figure 12.24 the resultant output Sangam graph `G'`.

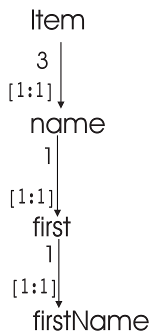


Figure 12.22: Fragment of the Input Sangam graph `G`.

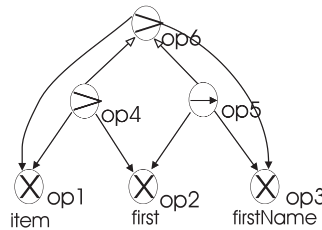


Figure 12.23: The Cross Algebra Graph Produced by the Inlining the Path `p:e1.e2.e3` as shown in Figure 12.22.

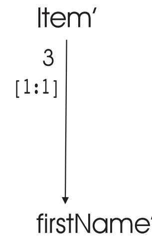


Figure 12.24: The Resultant Fragment of the the Output Sangam graph `G'` Produced by the Evaluation of the CAT in Figure 12.23.

- a recursive backpointer edge is handled by using the $\ominus$ operator to create a backpointer edge from the output (the parent) of the $\otimes$ operator to the output of the $\otimes$ operator that maps the recursive child node. Context dependency edges are then built from the $\ominus$ operator to the two $\otimes$ operators.

The evaluation of the CAG that is built by the above process will result in an output Sangam graph that can now be translated into a relational schema. As a Sangam graph can handle both set-valued

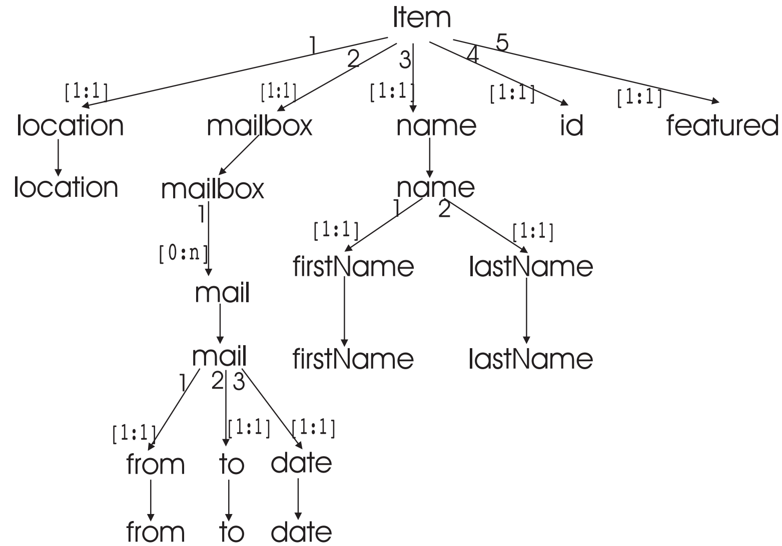


Figure 12.25: A Fragment of the XMark Benchmark DTD as shown in Figure 10.2 Depicted as a Sangam Graph (Sangam graph). No order and quantifier annotations are shown for the backpointer edges to distinguish them in the figure. These are defaulted to `1` and `[1:1] respectively.`

attributes as well as recursion, we do not have to handle these when building the cross algebra graph (CAG). Instead, the issue of translating the set-valued attributes and recursion is handled when a Sangam graph is translated into a relational schema as discussed in Section 11.2.2. Figure 12.26 graphically depicts the CAG that translates the Sangam graph in Figure 12.25 to the Sangam graph shown in Figure 12.27.

The Sangam graph in Figure 12.27 can now be translated based on the steps given in Section 11.2.2 to a relational schema is as shown in Figure 12.28.

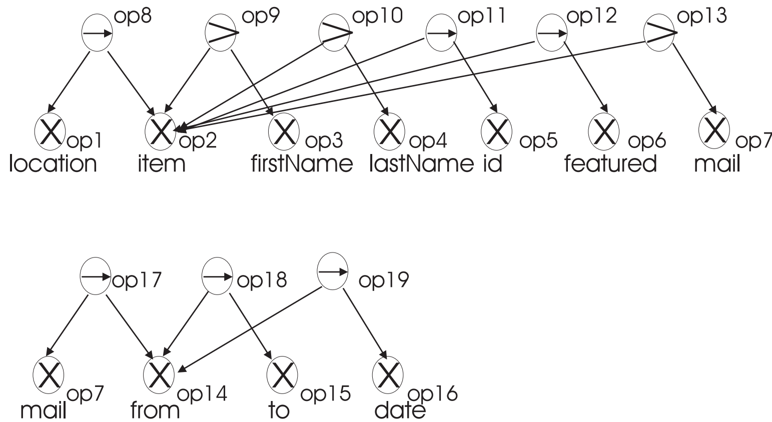


Figure 12.26: The Cross Algebra Graph that represents the Basic Inlining Technique applied to the Sangam graph in Figure 11.3. This Sangam graph is only for the node with root `item`. Cross Algebra Trees similar to the ones given in this figure will be produced for each root node.

12.4.2 The Shared Inlining

Shared Inlining Technique. The principal idea behind shared inlining is to identify the elements that would be represented by multiple relations by the basic inlining technique, and then to share them by creating separate relations for these elements. In this technique, relations are created for all elements in the DTD graph that have an in-degree greater than one. Nodes with in-degree of one are inlined and the nodes with in-degree of zero are made into separate relations. Mutually recursive elements are handled the same as in the basic inlining approach.

CAG for Shared Inlining Technique. The algorithm for creating a CAG for the shared inlining technique is similar to that of creating the CAG for a basic inlining with a few differences. The main difference is that in the

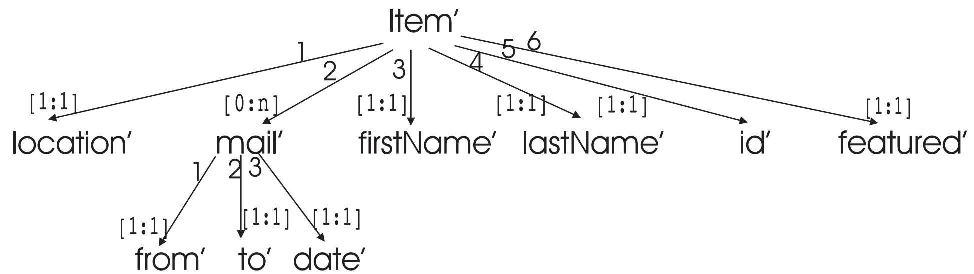


Figure 12.27: Output Sangam graph Produced by the Evaluation of the Cross Algebra Graph in Figure 12.26 for Input Sangam graph in Figure 12.25.

Item'(id' VARCHAR(10) NOTNULL, featured' VARCHAR(100) NOTNULL, location' VARCHAR(200) NOTNULL, firstName' VARCHAR(200), lastName' VARCHAR(200))

Item.mail'(from' VARCHAR(50), to' VARCHAR(50), date' VARCHAR(8), CONSTRAINT fk_id FOREIGN KEY (id') REFERENCES Item'(id'))

Figure 12.28: The Relational Schema Produced by the Translation of the Sangam graph in Figure 12.27.

CAG for basic inlining all nodes are mapped to the output via $\otimes$ operators. For example, the node item and location are mapped and produce output nodes item' and location'. A $\ominus$ operator is then used to re-create the edge e_1 between item and location to now exist between the nodes item' and location'. In addition all the sub-elements of the node location obtained by following the backpointer edge are inlined as direct children of the node location'. In addition, the node location representing a possible root is also mapped as such to the output. In shared inlining, this root node location is mapped to the output only if the number of backpointer edges for this node is greater than one, or if they are zero. All nodes with only one backpointer edge are inlined and no separate cross algebra tree is created to map it to the output. The rest of the algorithm remains the same as the basic inlining algorithm.

12.5 Summary

In this chapter, we have presented the basic transformation components of the Sangam middle-layer, and shown how a local application schema can be mapped from one data model to a schema in another data model via the Sangam middle-layer. To enable this transformation across data models, in this chapter we have defined (1) the primitive set of cross algebra operators that conform to the set of linear transformation operators [GY98]. These operators have been shown in graph transformation literature [GY98] to be sufficiently powerful to express complex transformations, and to be sufficient to express all linear transformations; (2) two different techniques of

composing algebra expression for these operators; (3) the physical binding and the evaluation algorithm for these operators; and (4) a concrete example that transforms the XMark [SWK+01] benchmark schema to a relational schema via the basic and shared inlining methods [STZ+99]. These techniques are represented by a cross algebra expression in the Sangam middle layer.

Our work on cross-algebra as a generic mapping language has significant advantages.

- *Re-usable transformations.* With the cross algebra, we are able to define many basic and complex transformations such as, the inlining of an XML schema into a relational schema. This cross algebra expression is not bound to any particular XML schema or relational schema, but rather can now be re-used to perform the inlining transformation between any two Sangam graphs.

- *Development of Re-usable tools.* Tools such as propagation algorithms, optimization strategies can now be developed based on the cross algebra expressions. These tools are re-usable for any cross algebra expression defined between schemas for any data model.

In the next Chapter, we provide the description of one such tool, the propagation algorithm, that can propagate change from the XML input to the relational output via the Sangam integration framework.

Chapter 13

Sangam: Evaluating the Theory

Based on the concepts of the Sangam graph model and the Cross Algebra Graph presented in Chapters 11 and 7 respectively, we have developed a middle-layer framework that can translate XML DTDs and documents to relational schemas and data via cross algebra graphs that can represent a variety of translation techniques. In this chapter, we present the overall architecture of the prototype system Sangam as well as a performance calibration of the various components of the Sangam architecture.

13.1 Architectural Overview of Cross Algebra Processing Engine

Figure 13.1 depicts the architecture and the data flow of our prototype system, Sangam. This system has been developed using Java technology and a variety of tools such as the JAXP [Sys01] for parsing XML documents and the DTD-Parser [Wut01] for parsing the DTDs.

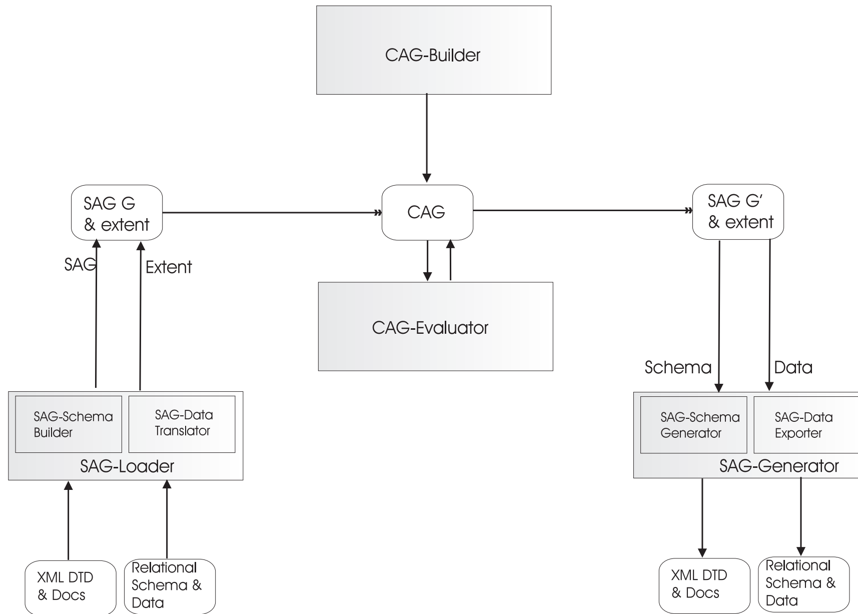


Figure 13.1: A Cross Algebra Framework for Relational and XML Models.

SAG-Loader: This module has two components. The first component *SAG-SchemaBuilder* provides APIs for loading and translating XML DTDs and relational schemas into SAGs. These are as per the algorithms given in Chapter 11. The second component is the *SAG-DataTranslator*. This component primarily translates XML documents and relational data. Based on the input, it creates new objects and inserts them into the extent of nodes and edges in a specified Sangam graph. The *SAG-DataTranslator* provides APIs to scan and load the data from a XML document or a relational table on a per node/per edge basis, i.e., its APIs allow for incremental loading given the object o needed to formulate the tuples. This allows materialization of the Sangam graph at run-time on a when needed basis. If however both

components are invoked during the initial load, then a fully materialized Sangam graph will be generated.

CAG-Builder: This module is responsible for building a cross algebra graph based on either the basic inlining or shared inlining technique and a given input Sangam graph. These CAGs are built as per the algorithms outlined in Section 12.4. Thus, the CAG-Builder takes as input a Sangam graph, and a parameter that specifies either the basic inlining or the shared inlining techniques. The output of this module is a CAG for the specified input Sangam graph G.

CAG-Evaluator: The core of the system lies in the CAG-Evaluator. Once a CAG has been built by the CAG-Builder, this module is responsible for generating the actual physical plan, executing the plan and hence producing the output Sangam graph and its translated data. Physical algebra operators and the evaluation algorithm are a core part of this module. Details of these are given in Chapter 12.3.

SAG-Generator: The SAG-Generator is responsible for generating the actual application schemas and for translating the Sangam graph data into a specific data model. As in the SAG-Loader module, there are two main components to this module: *SAG-SchemaGenerator* and *SAG-DataExporter*. The module *SAG-SchemaGenerator* can take a given Sangam graph and generate either an XML DTD or a relational schema as output. This is a user-specific choice and is specified at run-time via a parameter. The algorithm

for this is as described in Chapter 11. The *SAG-DataExporter* does the equivalent for the data, i.e., it translates the objects of the Sangam graph nodes and edges into either XML documents or relational extents.

13.2 Experimental Validation of Sangam

We have conducted several experiments to: (1) verify the feasibility of the Sangam graph model and the cross algebra graph as presented in Chapters 11 and 7 respectively; and (2) measure the costs for the different components of the system architecture (Section 13.1) used to transform one XML document to another XML document, or to transform one XML document to a relational database. Towards these two goals, we have conducted experiments to measure the performance of:

- the loading of the DTD and XML documents into a Sangam graph (Section 13.2.1);
- the generation of a DTD and associated XML documents, or a relational database from an output Sangam graph and its corresponding extent (Section 13.2.1);
- the evaluation of the CAG in terms of transformation execution (Section 13.2.2).

All experiments used as input the `auction.dtd`, available as part of the XMark Benchmark for XML [SWK+01], the `personal.dtd` and the `play.dtd`, available as a sample included in the JAXP 1.1 distribution. The

XML documents corresponding to the `auction.dtd` were generated using *xmlgen*, available for download from XMark [SWK+01]. The *xmlgen* tool however can generate XML documents corresponding to the `auction.dtd` only. Hence, to generate XML documents corresponding to `personal.dtd`, we wrote our own tool, called *xmlPGen*. A feature of the *xmlPGen* tool was that it allowed us to produce a uniform distribution of the number of objects for all nodes and edges of the Sangam graph. The `auction.dtd` and the `personal.dtd` are given in Appendix A.

For each input Sangam graph, we built (using the CAG-Builder) the Cross Algebra Graphs (CAGs) that represent either (1) an *identical*, called *ident*, transformation, in which case the input Sangam graph is identical to the output Sangam graph; or (2) a *basic inline*, called *inline*, transformation that models the basic inline technique presented by Shanmugasundaram et al. [STZ+99]; or (3) a *shared inline* transformation that models the shared inline technique [STZ+99][1].

All experiments were conducted on a Pentium IV, 933MHz, 256Mb RAM system running Debian Linux, kernel version 2.2.19, using the Sangam prototype system described in Section 13.1. The Sangam system itself was built using the Java JDK, version 1.3.1. To store generated relational schemas and data we used Oracle 9i running on Debian Linux, kernel version 2.2.19.

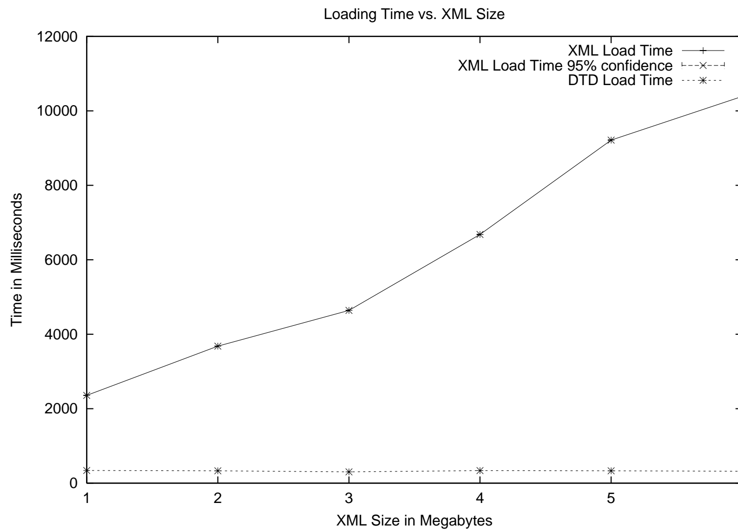


Figure 13.2: Time Taken to Load the `auction.dtd` and `auction.xml` Documents of Varying Sizes Into the Sangam Graph.

13.2.1 Loading to and Generating from Sangam Graphs

Figure 13.2 depicts the time taken to translate a given DTD and load its corresponding XML documents to a Sangam graph. We kept the DTD `auction.dtd` constant, and increased the size of the XML documents from 1MB to 7MB. Each data point represents the average of ten runs, shown with `95%` confidence intervals. As shown in the figure, the load times of the DTD itself are a small fraction of the time required to load an XML document. As seen by the slope of the XML load time, an increase in the size of the XML document causes a linear increase in the time taken to load the XML document into the Sangam graph. The **SAG-DataTranslator**

[1]We found the loading time of the basic and shared inlining to be almost identical. We thus report only the results of the basic inlining.

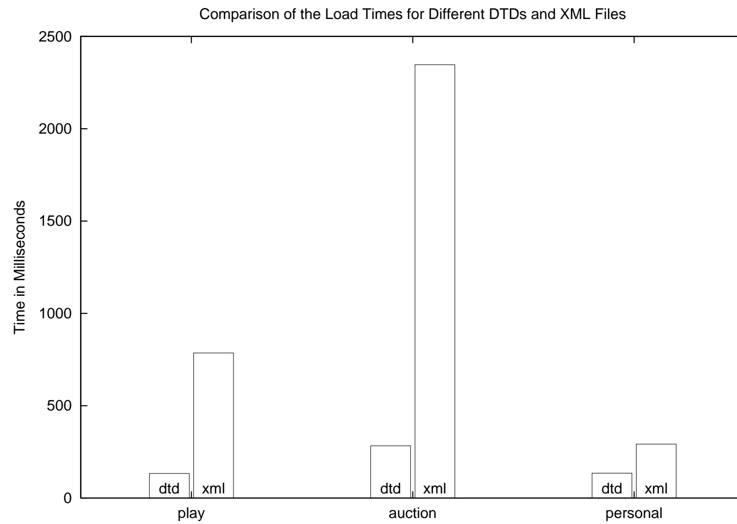


Figure 13.3: Comparing the Load Times for Different DTDs and Corresponding XML Documents.

algorithm creates one nodeObject per XML element, and a fixed number of edgeObjects representing the cardinality of the relationship between the element and the sub-elements (or its attributes).

Figure 13.3 depicts a bar-chart graph showing the different load times for three different DTDs, `personal.dtd`, `auction.dtd` and `play.dtd` and their corresponding XML documents `personal.xml`, `auction.xml`, and `play.xml` respectively. The `auction.dtd` was the largest with a size of `4k` bytes. The `personal.dtd` and `play.dtd` were `0.5K` and `1K` respectively. The `auction.xml`, `personal.xml` and `play.xml` were `3MB`, `1MB` and `27K` respectively. These times were consistent with our hypothesis that the loading times increase with the increase in size of the DTD and the XML documents.

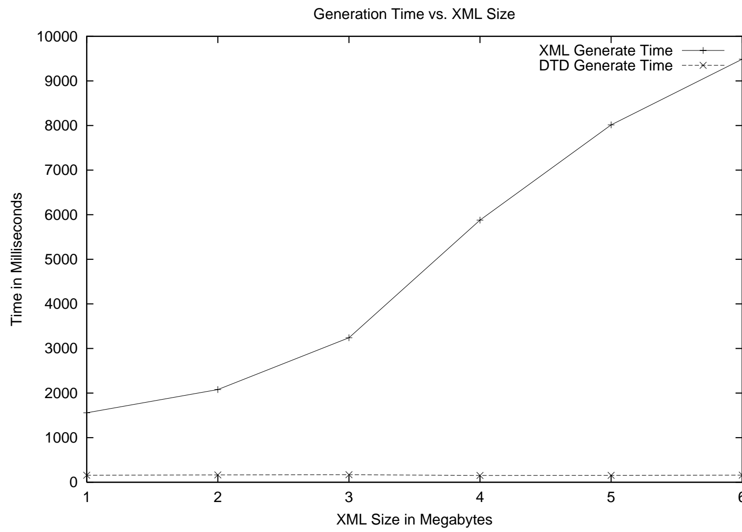


Figure 13.4: Time Taken to Generate the `auction.dtd` and Different `auction.xml` Documents of Varying Sizes From a Given Sangam Graph.

Figure 13.4 depicts the time to generate DTD and XML documents from a given Sangam graph. The Sangam graph structure used for the generation was initially loaded from `auction.dtd` and was kept constant. We varied the extent size of the nodes and edges in the Sangam graph. The different extent sizes for the Sangam graph were obtained by loading XML documents of varying sizes, from 1MB to 7MB, as the extent of the Sangam graph. To generate the DTD and XML documents, we performed a depth first traversal of the Sangam graph and loaded appropriate objects of the DTDParser [Wut01] and `javax.xml.transform.Transformer` provided as part of the JAXP 1.1 [Sys01] package, respectively. The actual generation of the DTD and the XML documents was done via the DTD-Parser [Wut01] and the JAXP 1.1 Transformer [Sys01]. In this experiment,

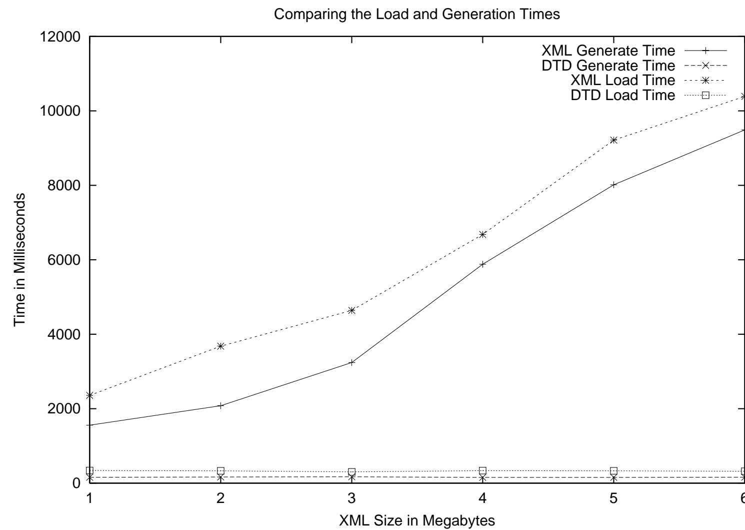


Figure 13.5: Comparison of the Time Taken to Load and Generate the `auction.dtd` and Different `auction.xml` Documents of Varying Sizes to and from a Sangam Graph.

we report the total time for the generation of the DTD and the XML documents, including the DTDParser generation and the JAXP generation. As shown in the figure, the generation times for the DTD are a small fraction of the time required to generate an XML document. However, these generation times are considerably smaller than the load values. The difference between the load and generate times can be attributed to the fact that the generation algorithms require only one-pass through the Sangam graph, as opposed to two passes need during the load process. Figure 13.5 compares the time taken to load and generate the `auction.dtd` and `auction.xml` to and from the Sangam graph.

13.2.2 CAG Evaluation

Performance of Cross Algebra Operators

Figure 13.6 depicts the time taken to evaluate one `Cross` algebra operator. We kept the operator `Cross` and its input node constant, and increased the extent of the input node from `1` to `20,000` objects. Each data point represents the average of ten runs. The evaluation time of a `Cross` operator increases linearly with a linear increase in the extent size of the input node.

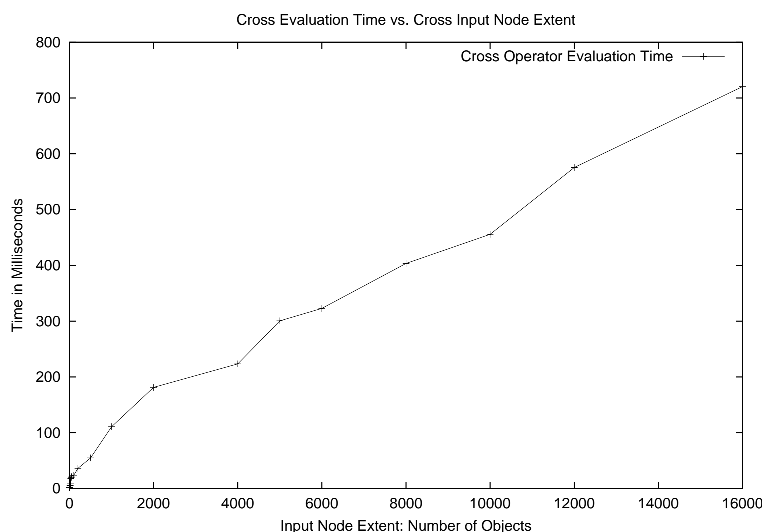


Figure 13.6: The Time Taken to Evaluate a Cross Algebra Operator as the Extent Size of its Input is Varied.

Figure 13.7 depicts the time taken to evaluate one `Connect` algebra operator. We kept the operator `Connect` and its input edge constant, and increased the extent of the input edge from `1` to `20,000` edge objects. Each data point represents the average of ten runs. The data in Figure 13.7 can

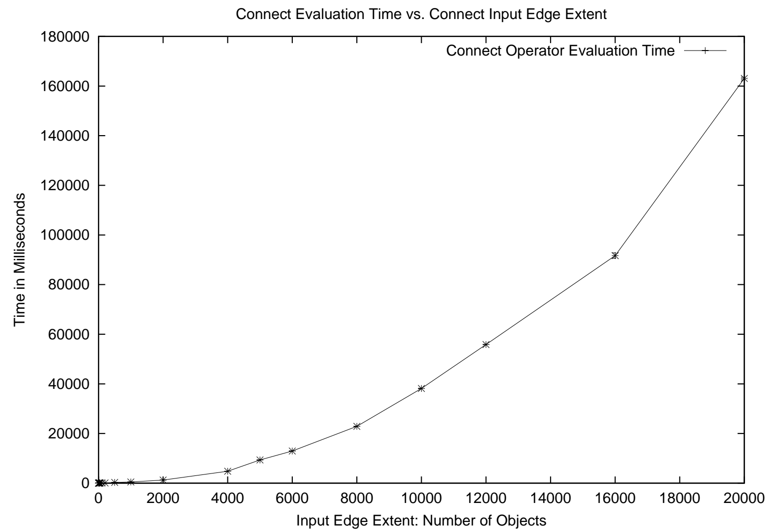


Figure 13.7: The Time Taken to Evaluate a Connect Algebra Operator as the Extent Size of its Input is Varied.

be well represented by a polynomial[2] with degree 2 with a coefficient of determination[3] of .99. Construction of one object of a `connect` operator requires searching through the objects of the two nodes that represent the end-points of the output edge. The evaluation time for the `connect` operator is thus $O(n^2)$. This is supported by our experimental results.

Figure 13.8 depicts the time taken to evaluate one `Smooth` operator. We kept the `Smooth` operator and its two input edges constant, and uniformly increased the extent of both its input edges from `1` to `20,000` edge objects. Each data point represents the average of ten runs, shown with `95%`

[2]The exact polynomial function is: $0.0004x^2 - 0.6x + 300$.

[3]The coefficient of determination, also known as the R-squared value, is an indicator that ranges in value from 0 to 1 and reveals how closely the estimated values for the trend line correspond to the actual data. A trend line is most reliable when its coefficient of determination is at or near 1.

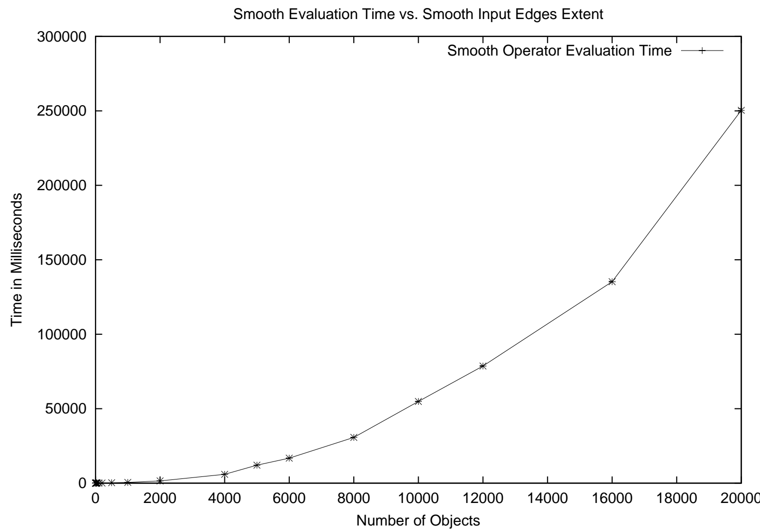


Figure 13.8: The Time Taken to Evaluate a Smooth Algebra Operator as the Extent Size of its Inputs is Varied.

confidence intervals. The data in Figure 13.8 can be well represented by a polynomial[4] with degree 3 with a coefficient of determination of .998. Construction of one object of a `smooth` operator requires searching through the objects of the first node, determining all objects of the second node to which it is connected, and then for each object of the second node determining all the objects of the third node to which it is connected, leading to evaluation times $O(n^3)$. This analysis for the evaluation times for the `smooth` operator is supported by our experimental results.

Figure 13.9 depicts the time taken to evaluate one `Subdivide` operator. We kept the `Subdivide` operator and its input edge constant, and increased uniformly the extent of its input edge from `1` to `20,000` edge

[4]The exact polynomial function is: $2e^{-0.08}x^3 + 0.0002x^2 + 1.1523x - 283.81$.

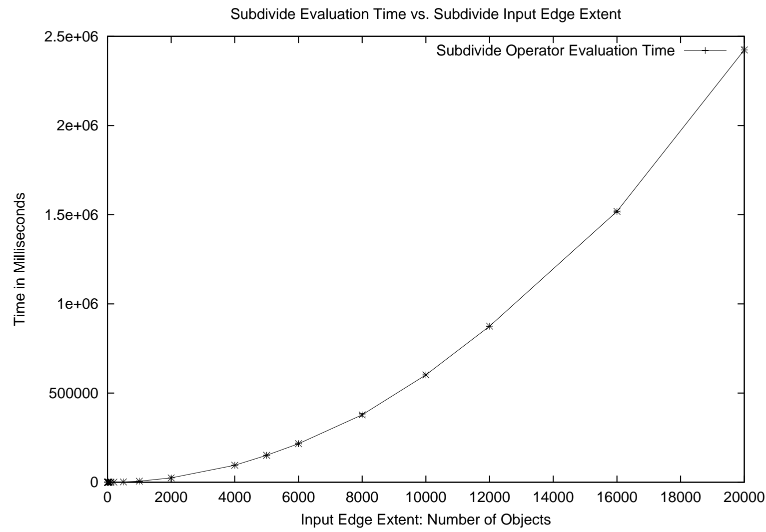


Figure 13.9: The Time Taken to Evaluate a Subdivide Algebra Operator as the Extent Size of its Input is Varied.

objects. Each data point represents the average of ten runs, shown with 95% confidence intervals. The data in Figure 13.9 can be well represented by a polynomial[5] of degree 2 with a coefficient of determination of .9999. The `subdivide` operator creates one nodeObject and two edgeObjects in the output for every edgeObject in the input. The search to locate the two endpoints is $O(n^2)$. However, this algorithm is dominated by the cost of creating the objects thereby explaining its faster growth.

Figure 13.10 summarizes the performance of the four algebra operators.

[5]The exact polynomial function is: $0.0061x^2 - 0.9863x + 571.38$.

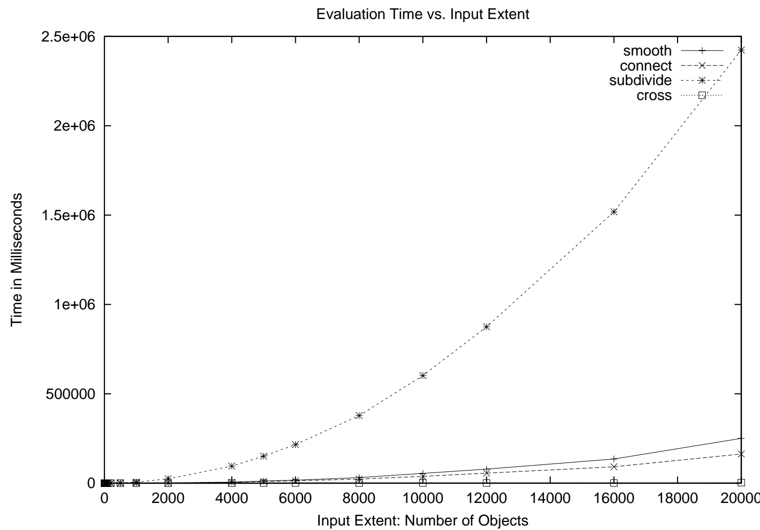


Figure 13.10: Performance Comparison of the Four Algebra Operators.

Performance of Inline and Ident CAGs

The **CAG-Builder** module can build *ident* and *inline* CAGs for a specified input Sangam graph. The evaluation of the CAG is dependent on: (1) the extent size of each node and edge in the input Sangam graph; and (2) the types of cross algebra operators that are part of the CAG.

Figure 13.11 depicts the evaluation time for the *ident* and the *inline* CAGs. Both the CAGs were built using the `personal.dtd` as the input. For each type of CAG, *inline* or *ident*, we kept the CAG and the input Sangam graph structure constant, and uniformly increased the extent size of the Sangam graph. Each data point represents the average of ten runs. The data for the `ident` CAG in Figure 13.11 can be well represented by a polynomial[6] of degree 2 with a coefficient of determination of .9989. This can be attributed

[6]The exact polynomial function is: $0.0037x^2 + 0.4836x - 924.85$.

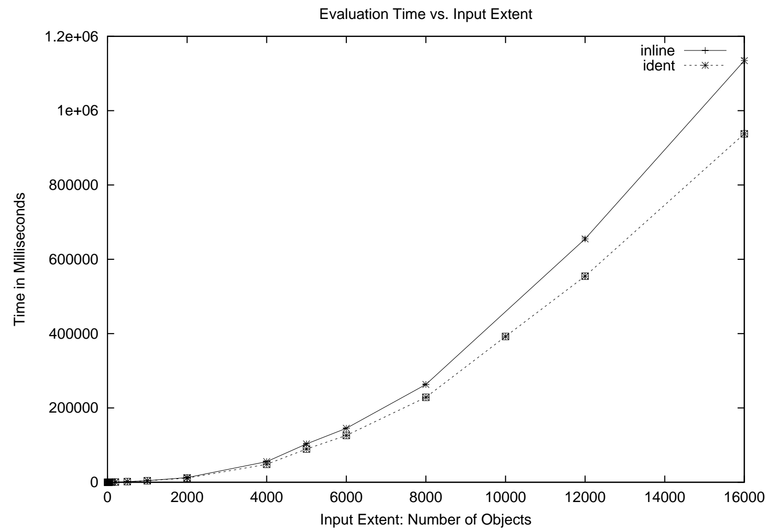


Figure 13.11: The Time Taken to Evaluate `Ident` and `Inline` CAGs as the Extent Size of the Input Sangam Graph Loaded from the `personal.dtd` is Varied.

to the fact that the ***ident*** CAG is comprised only of `cross` and `connect` algebra operators which are $\emptyset(\mathrm{n})$ and $\emptyset(\mathrm{n}^2)$ respectively. On the other hand, the performance of the `inline` CAG which comprises of `cross`, `connect` and `smooth` can be well represented by a polynomial[7] of degree 3 with a coefficient of determination of 0.9997. The dominant factor here is the performance of the `smooth` operator which is $\emptyset(\mathtt{n}^3)$.

Figure 13.12 also depicts similar results for the evaluation times of both the ***ident*** and the ***inline*** CAGs when run with the `auction.dtd` as input.

[7]The exact polynomial function is: $-5e^{-0.08}x^3 + 0.0058x^2 - 8.3567x + 1400.6$.

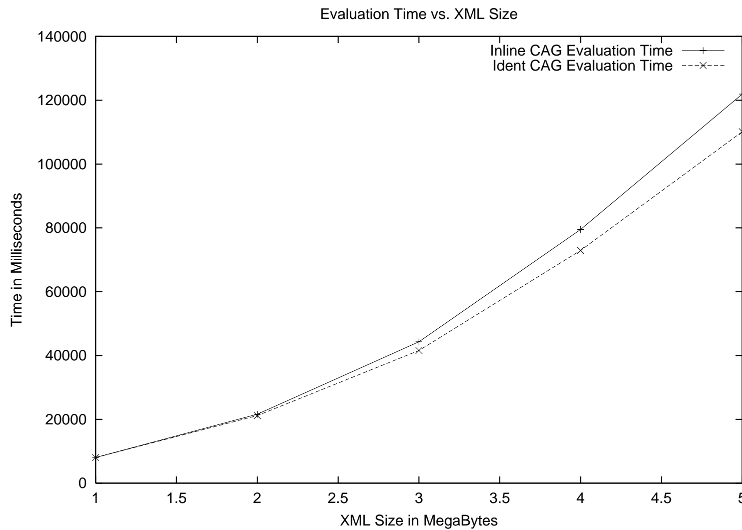


Figure 13.12: The Time Taken to Evaluate `Ident` and `Inline` CAGs as the Extent Size of the Input Sangam Graph Loaded from the `auction.dtd` is Varied.

13.2.3 Summary of Experimental Results

In summary, we can draw the following conclusions based on the experimental results:

- the DTD loading cost, i.e., the time taken to construct a Sangam graph from a given DTD, increases linearly with a linear increase in DTD size;
- the XML loading cost, i.e., the time taken to load the XML documents as the extent of the Sangam graph, increases linearly with a linear increase in XML size;
- the evaluation times for the `cross` operator increase linearly with a

linear increase in input extent size;

- the evaluation time for the `connect`, `smooth` and `subdivide` operators can be approximated by polynomials with respect to input extent size;
- the evaluation time for a CAG is a function of operator types and size of the CAG;

13.3 Summary

In this chapter, we present 𝔖𝔞𝔫𝔤𝔞𝔪, a prototype system that we have implemented based on the theory presented in Chapters 11 and 12. We have also presented a set of experiments that verify the feasibility of the Sangam graph model and the cross algebra graph as presented in Chapters 11 and 7 respectively; and also measure the costs for the different components of the system architecture (Section 13.1) used to transform one XML document to another XML document, or to transform one XML document to a relational database.

Chapter 14

Updating the Sangam Graph

Information is not static, rather it changes over time. A Sangam Graph represents an application (data model specific) schema and its data. These two are referred to as the local schema and local data. Any change in the local schema and data must be reflected in the Sangam graph . To enable this capability, we first present a taxonomy of primitive operations that can be applied on the Sangam graph resulting in a modified Sangam graph . This taxonomy of primitives includes operators for schema change to the Sangam graph (called SAG-SC, SAG-**S**chema **C**hange) as well as operators for data modification (called SAG-DU, SAG-**D**ata **U**pdate). As a second step we show how local schema and data changes expressed in the native data model data manipulation and data definition language can be modeled by a combination of SAG-SC and SAG-DU operations. Specifically we show how XML and relational updates can be translated into a sequence of SAG-SC and SAG-DU operations.

14.1 Schema Change Operations on the Sangam Graph

14.1.1 SAG-SC: Schema Change Primitives

The insertNode Operation. Given an input Sangam graph G such that node $m \in G$, the operation insertNode (ln, τ, m, el, q), creates a new node n[1] with type $\tau \in \Gamma$ and labelln, and inserts it into the input Sangam graph G to produce a new modified Sangam graph G′ such that a new edge e with label el and quantifier annotation q, connects the node n to the node m, i.e., e:<m, n>. The insertNode operator appends the node n as the last child of the node m[2], i.e., the edge e:<m, n> is inserted such that the order annotation $\rho(e) = j + 1$, where $j = max(\rho(e_i) | \forall\ e_i{:}<m, n_i>)$.

To handle order-specific insertions, such as with XML schemas, we define an additional insert operation insertNodeAt (ln, τ, m, el, q, pos). The insertNodeAt operation creates the new node n with the label ln and a new edge e:<m, n> with order annotation of pos, i.e., $\rho(e) = pos$. For all edges $e_i{:}<m,n_i> \in G$ such that the order annotation $\rho(e_i) \geq pos$, the order annotation of e_i is updated such that $\rho(e_i) = \rho(e_i) + 1$. The label and the quantifier annotation of the edge are specified by parameters el and q as before.

The insertNode operation fails if either (1) the label ln of the new node n is the same as the label l_i of some other node n_i and the nodes n and n_i share a common parent m; or (2) the type of the node n $\tau \notin \Gamma$. In addition to the above conditions the insertNodeAt fails if either (1) $\rho(e)$ is specified

[1] Recall from Section 11 that a node n has a label ln and an extent denoted as I (n).

[2] The node m maybe specified as null, in which case the node n is regarded as the root of the Sangam graph .

such that there exists no other edge e_i:<m, n_i> such that $\rho(e_i) < pos$; or (2) if $pos \leq 0$;

The deleteNode Operation. Given an input Sangam graph G, the operation deleteNode (n, m) removes the node n and the edge e:<m, n> to produce a new Sangam graph G'. For all edges e_i:<m,n_i> with order annotation $\rho(e_i) > \rho(e)$, the new order annotation of edge e_i is updated such that $\rho(e_i) = \rho(e_i) - 1$. For example, if node m connects to three nodes o, n, p with order annotations 1, 2 and 3 respectively, then the deletion of the node n results in node m containing two nodes o, p with order annotations 1 and 2 respectively.

The operation fails if either (1) there is no edge e:<m, n> in G; or (2) if the node n has other incoming edges (besides e); or (3) if it is not a leaf[3], i.e., is has other children nodes.

The insertEdge Operation. Given an input Sangam graph G, the operation insertEdge (m, n, l, q), creates a new edge e:<m, n> between the nodes m and n, with m, n $\in$ Sangam graph G, with label l and quantifier annotation q. Here the order annotation $\rho(e) = j + 1$, where $j = max(\rho(e_i) | \forall\, e_i : \langle m, n_i \rangle, n_i \in G)$. We also provide the insertEdgeAt operation, insertEdgeAt (m, n, l, q, pos), that inserts an edge e:<m, n> with label l and quantifier annotation q, such that the order annotation $\rho(e) = pos$. If there already exists an edge e_i:<m,n_i> such that $\rho(e_i) \geq pos$, then the order annotation of e_i is updated such that $\rho(e_i) = \rho(e_i) + 1$.

[3]For precise definition of a leaf, please refer to Chapter 11.

The insertEdge operation fails if either of the nodes n or m do not exist in the Sangam graph G. In addition to the above condition, the insertEdgeAt operation fails if either (1) pos is specified such that there exists no other edge e_i:<m, n_i> such that $\rho(e_i) < pos$; or (2) if $pos \leq 0$.

The deleteEdge Operation. Given an input Sangam graph G, the operation deleteEdge (m,l) deletes the edge e:<m, n> with label l from node m to node n. If there exists an edge e_i:<m,n_i> such that $\rho(e_i) > \rho(e)$, then the order annotation of e_i is updated such that $\rho(e_i) = \rho(e_i) - 1$.

The operation fails if either (1) there is no edge e:<m,n> $\in$ G with label l; or (2) if node n $\in$ G′ has no incoming edges, i.e., if the node n would no longer be connected in G′.

The rename Operation. The rename operation, rename (n, l′), modifies the label of the node n $\in$ G to label l′ to produce a new Sangam graph G′.

The operation fails if (1) the node n does not exist in the Sangam graph G; (2) if the label l′ is the same as the old label l; or (2) there exists a node n_i with label l′ such that both n and n_i have a common parent m in G′.

14.1.2 Completeness of SAG-SC Operations

Table 14.1 summarizes the taxonomy of change operations that we have presented above. This taxonomy intuitively captures all changes needed to manipulate the Sangam graph (refer Section 11). Here we outline a proof that shows that this set indeed subsumes every possible type of graph change (completeness criteria). The proof given here is based on the com-

pleteness proof given by Banerjee et al. for the evolution taxonomy of Orion [BKKK87].

SAG Primitive	Description
insertNode (ln, τ, m, l, q)	Creates new node with label ln and inserts it as child of node m
insertNodeAt (ln, τ, m, l, q, pos)	Creates new node with label ln and inserts it as child of node m at position pos
deleteNode (m, n)	Deletes child node n from parent m
insertEdge (m, n,l, q)	Inserts new edge between nodes m and n, making n child of node m
insertEdgeAt (m, n,l, q, pos)	Inserts new edge at position pos between nodes m and n, making n child of node m
deleteEdge (m, l)	Deletes edge e with label l from the node m
rename (n, l')	Modifies label of node n to l'

Table 14.1: Taxonomy of Sangam Graph Structural Change Primitives.

In order to show completeness of these operations we prove that every valid Sangam graph as per Definition 6 can be generated by the operations in Section 14.1 from any other given Sangam graph .

Lemma 9 *For any given Sangam graph* G*, there is a finite sequence of deleteNode operations that can reduce the Sangam graph* G *to a Sangam graph* G' *consisting of only one single node.*

Proof: It is apparent that if we repeatedly apply the operation deleteNode to a leaf node n which removes a node n and an edge e:<m, n> from a Sangam graph G, we can after a finite number of applications reduce any given Sangam graph G to a new Sangam graph G' which only has one node.

Lemma 10 *There is a finite sequence of operations* {*insertNode , insertNodeAt* } *that generates any desired Sangam graph* G *from an empty Sangam graph* G'*.*

Proof: Let G be a Sangam graph with a finite number of nodes and edges and one root, and G′ be an empty. The following procedure can, via the finite sequence of operations listed above, transform the empty Sangam graph G′ to any desired Sangam graph G. To achieve this, we traverse G in a breadth-first order starting at the root node r of G and do the following:

1. Set node n $\leftarrow$ r.
2. For the node n, add a corresponding node n′ to Sangam graph G′ using operation insertNodeAt (n.label, n.τ, n.parent, ne.label, Ω(ne), ρ(ne)), where ne: <n.parent, n>. The node n′ has the same label as n and the same type τ.
3. Traverse the set of outgoing edges from the node n and for each outgoing edge e:<n, r>, repeat steps (1), (2) and (3) for r.

The resultant Sangam graph G′ is equivalent to the initial Sangam-graph G as they have the same set of nodes and edges but different OIDs. □

Theorem 9 *Given two arbitrary Sangam graphs* G *and* G′*, there is a finite sequence* F *of operations of type* { *insertNode* , *insertNodeAt* , *deleteNode* }*, such that when* F *is applied to the Sangam graph* G *it produces the Sangam graph* G′*.*

Proof: We can prove this by first reducing the Sangam graph G to an intermediate Sangam graph G1 using Lemma 9. The Sangam graph G1 can then be converted to Sangam graph G′ using Lemma 10. □

The set of operations { insertNode , insertNodeAt , deleteNode } is a subset of the operations { insertNode , insertNodeAt , deleteNode , insertEdge ,

insertEdgeAt , deleteEdge , rename }. Hence the completeness of this set of operations follows from Theorem 9.

14.1.3 Correctness of Sangam graph -SC Operations

We next define correctness of a SAG-SC operation as given below.

Definition 23 (Correctness) *A SAG-SC operation* c *applied on a Sangam graph G is* correct*, if when given a valid Sangam graph G its application produces as output a Sangam graph G ' that also is valid by Definition 6.*

Theorem 10 (Correctness of SAG-SC Operations) *All operations* $c \in$ {*insertNode , insertNodeAt , deleteNode , insertEdge , insertEdgeAt , deleteEdge , rename* } *are* correct*.*

Proof 1 *Assume that the input Sangam graph is a valid Sangam graph by Definition 6. Now let us consider the deleteNode operation. We know by Definition 6, a valid Sangam graph must be (2) connected and (3) type τ of all nodes* n *in* G *is in Γ. As the deleteNode operation deletes a node, property (3) is not affected. In the deleteNode operation we observe that the operation fails if the node* n *that is to be deleted has any outgoing edges. This prevents the creation of un-connected Sangam graphs . Moreover, the deleteNode operation removes the edge* e *from the parent* m *to the child node* n*, such that no dangling edges without target nodes would remain. Thus, the deleteNode operation produces a valid Sangam graph as output when given a valid Sangam graph as input.*

Next consider the insertNode operation. Given that the input Sangam graph G *is valid, an insertNode operation inserts a new node with label* l *and type τ. The*

operation insertNode fails if $\tau \notin \Gamma$, ensuring that on completion of insertNode operation, all nodes of G' *have a type τ in Γ. The insertNode operation inserts a node* n *in* G' *as a child of a specified parent node* m*, thus preventing the creation of un-connected Sangam graphs . Moreover, the insertNode operation appends the node as a child of the parent node. Hence the order annotation of the outgoing edges from the parent is maintained.*

A similar inspection of the other SAG-SC operations shows that they produce valid Sangam graphs as output.

14.2 Data Modification Primitives for the Sangam Graph

Similar to the SAG-SC operations, we now define four data modification primitives that can insert and delete nodeObjects into and from the extent I (n) of a node n, and insert and delete edgeObjects into the extent R (e) of an edge e.

The addObject Operation. The addObject operation, given as addObject (v,n), creates a new object o with data value v and inserts the object o into the extent I (n) of node n.

The addObject operation fails if either (1) for any outgoing edge e of node n the minimum quantifier annotation is 1, requiring that the object o must be connected to some other object o_i in the extent I (n_i) of node n_i, where n_i is a child of node n; or (2)for any incoming edge e_m:<m, n>, the minimum quantifier annotation is 1, requiring that the object o must be

connected to some other object o_m in the extent I (m).

The deleteObject Operation. The deleteObject operation, given as deleteObject (o,n), removes the object o from the extent I (n) of node n.

The operation fails if (1) there is no object $o \in$ I (n); (2) if there exists any edgeObject o_e: $<o_1, o_2>$ in R (e) such that $o_1 = o$ or $o_2 = o$ and e is an outgoing or an incoming edge of node n.

The renameValue Operation. The renameValue operation, given as renameValue (o, v', n), updates the value of object $o \in$ I (n) to the new value v'.

The operation fails if there exists no object $o \in$ I (n).

The addEdgeObject Operation. The addEdgeObject operation, given as addEdgeObject (o_1,o_2,e), creates a new edgeObject o_e:$<o_1, o_2>$ and inserts the edgeObject o_e into the extent R (e) of edge e:$<$m, n$>$.

The operation fails if (1) $o_1 \notin$ I (m) or $o_2 \notin$ I (n); (2) there is no edge e:$<$m,n$>$ $\in$ G; (3) the edgeObject o_e:$<o_1,o_2>$ already exists, i.e., o_e:$<o_1,o_2>$ $\in$ R (e); or (4) the addition of the edgeObject o_e:$<o_1,o_2>$ violates the maximum quantifier annotation of the edge e.

The deleteEdgeObject Operation. The deleteEdgeObject operation, given as

deleteEdgeObject (o_e,e), removes the object o_e from the extent R (e) of edge e:m$\rightarrow$n.

The operation fails if (1) $o_e \notin$ R (e); (2) the deletion of o_e:<o_1, o_2> violates the minimum quantifier annotation of the edge e; or (3) the removal of the edgeObject o_e causes the nodeObject o_2 to become dis-connected.

14.3 Translation Completeness of Sangam Graph Operations

14.3.1 Translation Completeness of SAG-SC Operations

Perhaps more important than the completeness of the SAG-SC operations shown in Section 14.1.2 is the completeness of the SAG-SC operations with respect to local data model changes. We term this as the *translation completeness*. For example, if we consider the XML data model, then we call the SAG-SC operations *translation complete* if all XML schema changes can be translated into a sequence of SAG-SC operations that achieve the same effect on the Sangam graph that represents the XML DTD. In this section, we show that the SAG-SC operations are indeed *translation complete* with respect to XML and relational schema changes.

XML Changes

To execute schema changes on XML, we use the **XEM** system [SKC+01]. The **XEM** system (**X**ML **E**volution **M**anager) has been developed by Hong et al. at Worcester Polytechnic Institute is a complete XML evolution facility that provides a minimal yet complete taxonomy of XML evolution operations. Table 14.2 enumerates the DTD operations provided by XEM.

DTD Operation	Description
createDTDEl(*u*)	Create element with name *u*
destroyDTDEl(*u*)	Destroy element with name *u*
renameDTDEl(*u, u'*)	Rename element from name *u* to *u'*
insertDTDEl(*E, pos, P, q, d*)	Add element *E* at position *pos* to parent *P* with quantifier *q* and default value *d*
removeDTDEl(*E, P*)	Remove sub-element *E* in parent *P*
changeQuant(*E, P, q, d*)	Change quantifier of subElement `E` in parent *P* to quantifier *q*
convertToGroup(*E, start, end*)	Group sub-elements from position *start* to position *end* in parent *E* into a list group
flattenGroup(*E, pos*)	Flatten group at position *pos* in element *E* to a list of sub-elements
addDTDAtt(*u, E, t, d, v*)	Add attribute with name *u* to element *E* with type *t*, default type *d*, and default value *v*
destroyDTDAtt(*u, E*)	Destroy attribute with name *u* from element *E*
changeAttDefType(*u, E, t, v*)	Change element *E*'s attribute *u*'s type to *t*, with default value *v*
changeAttDefValue(*u, E, v*)	Change element *E*'s attribute *u*'s default value to *v*

Table 14.2: DTD Data Change Primitives of XEM [SKC$^+$01].

Now consider the operation `insertDTDEl(E, pos, P, q, d)` that adds element `E` at position `pos` to parent `P` with quantifier `q` and default value `d`. To translate this operation into an operation(s) on the Sangam-graph , we need to consider that[4]: (1) each XML element is represented by a node in the Sangam graph . Hence, the elements `E` and `P` are represented by Sangam graph nodes `n` and `m` respectively; (2) the parent-child relationship is represented by an edge `e` between the Sangam graph nodes `m` and `n`; (3) the quantifier `q` is translated into the quantifier annotation on edge `e`. The rules for the quantifier translation are as given in Section 11; (4) the order of the element `e` as given by the parameter `pos` is translated into the

[4]This is a summarization of the process to translate an XML DTD into a Sangam graph . For more details on this refer to Chapter 11.

order annotation for the edge e; and (5) the default value d is a property of element E and is represented by atomic node dv[5]. The node dv is connected to the parent node n by an edge de:<n, dv> that has a quantifier annotation of [1:1], and order annotation of 1.

Assuming that the XML elements P and E are represented by nodes m and n in Sangam graph G respectively, the insertDTDEl operation maps to the sequence of the following operations. Here the operation insert-DTDEl is inserting an element E as a subelement of the element P. The elements E and P are already defined in the DTD. We make the same assumption for the Sangam graph .

DTD Operation	SAG-SC Sequence	Description
createDTDEl(u)	insertNode (u, τ, D, l, q)	Creates a new node n with label u and inserts it as a child of node D such that there is now an edge e:<D, n>. The edge e has a label l and quantifier annotation of q. The order annotation, ρ(e) is as calculated as specified for operation insertNode .
destroyDTDEl(u)	deleteNode (n, D)	Deletes node n with label u from parent D
renameDTDEl(u, u′)	rename (n, u′)	Renames the label of node n from u to u′
insertDTDEl(E, pos, P, q, d)	insertNodeAt (n.label, τ, m, l, q′, pos)	Creates a new node n_s with label n.label where n is the node that represents the definition of element E. The node n_s is inserted as a child of node m where m represents the XML element P. The edge e:<m, n_s> has a label l, quantifier annotation Ω(e) =q and an order annotation ρ(e) = pos.
	insertEdge (n_s, n, l, q)	Creates an edge e_p:<n_s, n> with label l and Ω(e) = q.
	insertNode (dv, τ, n_s, "default", q)	Inserts an atomic node dv as a child of node n_s to model the default value
removeDTDEl(E, P)	deleteNode (m, n)	Deletes the node n, and edge e:<m, n>

Table 14.3: Translation of XEM Primitives to SAG-SC Operations.

1. insertNodeAt (n.label, τ, m, "childElement", q′, pos). This operation creates a new node n_s with the same label as the node n, and

[5]The default value is used when new nodeObejcts are created without a specific value.

DTD Operation	SAG-SC Sequence	Description
changeQuant(E, q, q')	insertEdge (m, n, e.label, q')	Inserts a new edge e':<m, n> with Ω(e') = q'. The label of e' is the same as label of e; and ρ(e') = ρ(e).
	deleteEdge (m, e.label)	delete edge e:<m, n>.
convertToGroup(E, start, end)	insertNodeAt (ln, τ, n, l, q, start)	Creates a new node g with label ln and inserts the node as a child of node n. The edge e:<n, g> is created between the nodes n and g with a label l, Ω(e) = [1:1] and ρ(e) = start.
	i $\leftarrow$ start + 1 while i $\neq$ end + 1 : n_i $\leftarrow$ nodeAt(n, i)	Retrieve the node n_i such that $\rho(e_i)$ = i for edge e_i:<n, n_i>
	pos $\leftarrow$ 1	Initialize the order index to 1
	insertEdgeAt (g, n_i, e_i.label, $\Omega(e_i)$, pos)	Insert edge e:<g, n_i> with the label, quantifier annotation and order annotation of the edge e_i:<n, n_i>
	deleteEdge (n, e_i.label)	Delete the edge e_i:<n,n_i>
	pos++	Increment the order variable

Table 14.4: Translation of XEM Primitives to SAG-SC Operations.

DTD Operation	SAG-SC Sequence	Description
flattenGroup(E, pos)	g $\leftarrow$ nodeAt(n, pos)	Get the node g such that ρ(e) = pos, where e:<n, g>
	children $\leftarrow$ children(g)	Get all the nodes n_i such that there exists an edge e_i:<g, n_i>
	for n_i in children:	
	insertEdgeAt (n, n_i, e_i.label, $\Omega(e_i)$, pos)	Create a new edge e_i':<n, n_i> such that label of e_i' is the same as label of e_i, $\Omega(e_i') = \Omega(e_i)$ and $\rho(e_i')$ = pos. For the first node n_i, $\rho(e_i') = \rho(e)$ = pos.
	deleteEdge (n, e.label)	Delete the edge e:<n, g>
	pos++	Increment the order variable
	deleteNode (n, g)	Delete the node g, a child of node n
addDTDAtt(u, E, t, d, v)	insertNode (u, n, l, q)	Create a node n_i with label u and insert it as a child of node n. For this create an edge e_i:<n, n_i> such that e_i has a label l, $\Omega(e_i)$ = q and $\rho(e_i)$ is assigned based on rules defined under the insertNode operation.
	for each property p:	For each XML property, (type, default, defaultType) insert atomic nodes that capture the schema level properties of the node
	insertNode (p, n_i, l, q)	Create a new atomic node an with label p and insert it as a child of node n_i such that the edge e_{an}:<n_i, an> has a label l, Ω(an) = q and $\rho(e_{an})$ is as per the operation insertNode .
destroyDTDAtt(u, E)	deleteNode (n, n_i)	Delete the node n_i with label u such that e:<n,n_i>

Table 14.5: Translation of XEM Primitives to SAG-SC Operations.

inserts the node as a child of node m. The edge e:<m, n_s> has a quantifier annotation q', an order annotation of pos and a label of "childElement". The quantifier annotation q' = [0:n] if input pa-

rameter q of the insertDTDEl operation is "$*$", q$'$ = [1 : n] for q = "+", q$'$ = [0 : 1] for q = "?", and q$'$ = [1 : 1] otherwise.

2. insertEdge (n_s, n, "subElement", q). This operation inserts an edge e from the newly created subelement n_s to its definition n. This definition is given by the node n. The quantifier q for this edge e is always set to [1 : 1], and the order annotation is set to 1.

3. insertNode (dv, τ, n_s, "default", q): This operation inserts a new atomic node dv to capture the default value of the element E when it occurs as a child of the element P. Here the τ indicates that this has an atomic node type. An edge e:<n_s,dv> is created and inserted as an outgoing edge of n. The quantifier q for the edge e:<n_s,dv> is set to [0 : 1].

The above sequence of updates simulates the insertion of the element E as a subelement of element P in an XML DTD. Tables 14.3, 14.4 and 14.5 give the synopsis of the update sequence of SAG-SC operations that achieve the semantics of the XEM operations. Here we assume that the Sangam graph node D represents the DTD node, the nodes m and n represent the elements P and E respectively, and unless specified otherwise τ(n) represents a complex Sangam graph node type.

Relational Schema Changes

For the relational model we consider the set of operations provided for schema evolution in SQL-99 [Tr00]. A set of common schema operations for commercial database systems is given in Table 14.6.

Evolution Primitive	Description
CREATE TABLE	Creates a new table
DROP TABLE	Drops the table
ALTER TABLE ADD COLUMN	Adds a column to the table
ALTER TABLE DROP COLUMN	Drops a column of the table
ALTER TABLE ALTER COLUMN <name>	Alters the column label
ALTER TABLE ALTER COLUMN SET DATA TYPE	Alters the column data type

Table 14.6: Taxonomy of Relational Schema Evolution Primitives.

Table 14.7 gives the sequence of SAG-SC operations for each of the relational schema evolution operations. Here we assume that the Sangam graph node `S` represents the entire relational schema, i.e., all relational tables for the schema are children of the node `S` (For more details please refer to Chapter 11). All relational evolution operations are assumed to operate within the schema `S`. For example, the operation CREATE TABLE `t` creates a new table `t` in the default schema which here is assumed to be schema `S`. As before τ always indicates a complex node unless specified otherwise.

14.3.2 Translation Completeness of SAG-DU Operations

Given that the goal of our work is to enable the translation of a schema and its data in one data model to another, it is essential that similar to the local schema changes, we are able to translate all data changes in one data model to data changes in another data model. We show in this section the translation completeness of the SAG-DU operations with respect to the XML and relational data updates.

Relational Primitive	SAG-SC Operations	Description
CREATE TABLE t	insertNode (t, τ, S, l, q)	Creates a node n with label t. It inserts the node n as a child of the node S such that e:<S, n>. The edge e has a label l and Ω(q) = q. Here q = [1:1]. The order annotation ρ(e) is assigned by the system as per the definition of the insertNode operation.
DROP TABLE t	deleteNode (n, S)	Deletes the node n with label t.
ALTER TABLE t ADD COLUMN c	insertNode (c, τ, n, l, q)	Adds a node n_i with label c. It inserts the node n_i as a child of node n such that there is an edge e:<n, n_i> with label l, Ω(e) = q. Here q = [0:1], if attribute can be NULL or q = [1:1] if attribute is NOT NULL. The order annotation ρ(e) is assigned by the system as per the definition of insertNode operation.
ALTER TABLE t DROP COLUMN c	deleteNode (n, n_i)	Deletes the node n_i with label c from the parent node n.
ALTER TABLE t ALTER COLUMN c <name>	rename (n_i, l')	Modifies the label of the node n_i from c to l'.
ALTER TABLE t ALTER COLUMN c SET DATA TYPE ty	rename (dt, ty')	Modifies the label of node dt that is a child node of the node n_i with label c to ty'.

Table 14.7: Translation of Relational Schema Evolution Primitives to SAG-SC Operations.

Translating XML Data Updates

We assume that all changes to XML documents are made via the the XML update primitives proposed by Tatarinov et al. [TIHW01]. Table 14.8[6] gives

[6]They do not make the separation between the value and the child in their work. Rather they are both part of the syntactical extension they propose for Quilt (XQuery). To

the four data update operations that we consider here. The operation sub-update [TIHW01] is not considered here. For more details on the XML update operations we refer the reader to [TIHW01].

XML Update Operation	Description
delete(`child`)	if the child is member of *target* object, it is removed. A child can be attribute, PCDATA and element
rename(`child`, `name`)	if the child is a non-PCDATA member of the *target* object, it is given the new `name`
insert(`content`)	inserts new content (which can be PCDATA, element or attribute) into the *target*
replace(`child`, `content`)	equivalent to insert(`content`) followed by delete(`content`)[7]

Table 14.8: XML Update Primitives

Consider the general update operation insert(`content`). An example of this operation is given in Figure 14.1 using XQuery extensions as proposed by Tatarinov [TIHW01]. Here the command INSERT ATTRIBUTES (`featured = ''true''`) inserts the value `''true''` for attribute `featured` where the target element is `item` with `ID=1` in the document `''item.xml''`.

```
FOR $it in document("item.xml")/ item[ID=''curio'']
        UPDATE $it {
            INSERT ATTRIBUTES (featured=''true'')
            INSERT <location>USA</location>)
}
```

Figure 14.1: An Example XQuery Statement to Insert Data Values into an XML Document.

To translate this operation into a sequence of SAG-DU operations, we assume without loss of generality that we have (1) a method `find` which

simplify we make this distinction clear.

returns the Sangam graph node representing a given schema entity label; (2) a method getNodeObject(I (n), v) that returns the set of nodeObjects $o_i \in$ I (n) that have a value v; and (3) method getEdgeObject(R (e), o_i) that returns the edgeObjects o_e:<o_1, o_2> $\in$ R (e) when either $o_1 = o_i$ or $o_2 = o_i$. To simplify the syntax used for the example, let the target item be represented by node m, the ID by node n, the attribute featured by node p, the edge mn:<item,ID> and the edge mn:<item,featured> in the Sangam graph that represents the XML schema (see Figure 11.3). The operation INSERT ATTRIBUTES (featured = ''true'') is translated to the following sequence of SAG-DU operations. Here we use the XQuery as shown in Figure 14.1 to provide the context/input parameters that may be needed for the translation to SAG-DU operations.

1. addObject (''true'',p): Creates a new object o with value ''true'' and inserts it into I (p). Recall that here node p represents the XML attribute featured.

2. The query in Figure 14.1 performs an update to the element item with ID = ''curio''. We assume here that the Sangam graph is materialized, and all item and ID objects have been created, translated from the XML document and inserted into the extents of nodes m and n in the Sangam graph respectively, as per the algorithms given in Section 11. Let $o_e \leftarrow$ getEdgeObject(R (mn), getNodeObject(I (n),curio) return the edgeObject o_e:<o_1, o_2> that links an object o_1 of node item to an object o_2 of node ID such that the value of o_2 = curio.

3. addEdgeObject (o_1, o, mp): Creates a new edgeObject o_{e2}:<o_1, o> and inserts it into the extent R (mn) of edge mp:<m, p>, where m represents the XML element item and p the XML attribute featured.

The result of the above sequence of steps is the creation of a new object that represents the statement featured = ''true'' that is now linked to the correct item object. Let r denote the Sangam graph node that represents the XML Element location, and the edge mr:<m, r> that represents the edge between the parent XML element item and the child element location. The command INSERT (<location>USA</location>) can then be translated to a sequence of SGM-DU operations as follows. Here we use the tag names in the query to find the right node into which the value is inserted.

1. addObject (''USA'', r): Creates a new object o_r with value ''USA'' and inserts it into I (r).
2. As the query in Figure 14.1 performs an update to the element item with ID = ''curio'', we again find the right object $o_i \in$ I (m) that is in relationship with the object o_r. This is the same as Step 2 above.
3. addEdgeObject (o_1, o_r, mr): Creates a new edgeObject o_{e3}:<o_1, o_r> and inserts it into the extent R (mr) of edge mr:<m, r>, where m represents the XML element item and r the XML element location.

In general, we translate the insert operations given as insert(content) [TIHW01] to a pair of addObject and addEdgeObject operations for every level of nesting. The

INSERT ATTRIBUTES (`featured=''true''`) operation has one level of nesting and hence often is translated to one pair of addObject and addEdgeObject functions. The INSERT (`<location>USA</location>`) operation also has one level of nesting and hence is translated to a pair of addObject and addEdgeObject operations.

Table 14.9 summarizes the translation of the XML data change primitives given in Figure 14.8 into SAG-DU operations. We assume that these methods are embedded in an XQuery statement that provides the context for these operations. For the Table 14.9 let `m` and `n` be Sangam graph nodes that represent XML elements and/or attribute. Let there be an edge `mn`:<`m`, `n`>. Assume that $o_c \in$ I (`n`) is a nodeObject such that `child` is the value of o_c. Let o_p be a nodeObject such that $o_p \in$ I (`m`) and o_p has a value `parent`.

Translating Relational Data Updates

For the relational data updates we consider the updates: `add-tuple`, `delete-tuple` and `modify-tuple`. Consider that we have a table `ITEM(ID NUMBER(4), featured VARCHAR(40))`. The Sangam graph `G` representing this schema will have a node `m` that represents the relation `ITEM`, nodes `i` and `f` that represent the `ID` and `featured` columns. The edges e_1 and e_2 represent the edge between `ITEM` and `ID`, and the edge between `ITEM` and `featured` respectively. The `add-tuple` data update operation given as `add-tuple` (“curio”, “true”) for the table `ITEM` is translated as follows:

XML Update Operation	SAG-DU Sequence	Description
delete(child)	deleteEdgeObject (o_e, e)	Based on the above assumption, first find the edgeObject o_e:<o_p, o_c> ∈ R (e) (provided by the XQuery context). Delete the edgeObject o_e.
	deleteObject (o_c, n)	Delete the object o_c with value child from the extent I (n) of node n
rename(child, name)	renameValue (o_c, name, n)	Renames the value child to name for object o_c ∈ I (n). The OID of the object remains the same
insert(content)	addObject (content, n)	Creates a new object o_c with value content. Adds new object o_c to I (n)[8]
	addEdgeObject (o_1, o_c, e)	If e represents the edge between the parent and the child nodes, then create new edgeObjects and add them to R (e).

Table 14.9: Translation of XML Update Operations to SAG-DU Operations.

1. Create a new nodeObject o_1 and insert into I (m) to represent a new row in the relational table.
2. Create a new nodeObject o_2 with value ''curio''. Insert this value into I (i).
3. Create a new edgeObject o_{e1}:<o_1, o_2> to represent the relationship between the object o_1 and o_2. Insert this object into R (e_1).
4. Create a new nodeObject o_3 with value ''true''. Insert this value into I (f).
5. Create a new edgeObject o_{e2}:<o_1, o_3> to represent the relationship

SAG Primitive	XEM Primitive	Description
insertNode (ln, τ, m, l, q)	createDTDEl(ln)	If m represents the DTD node.
	insertDTDEl(ln, ρ(e), m.label, q, d.label)	If n represents an element. Here e is an edge such that e:<m, n>. The node d represents a property node of the n.
	addDTDAtt(ln, m.label, t.label, d.label, v.label)	If node n represents an attribute. Here t, d, and v are property nodes that are children of node n. They may be empty.
insertNodeAt (ln, τ, m, l, q, pos)	insertDTDEl(ln, pos, m.label, q, d.label)	If n represents an element
	addDTDAtt(ln, m.label, t.label, d.label, v.label)	If node n represents an attribute. Order and Quantifier annotations are not used in the translation here.
deleteNode (m, n)	destroyDTDEl(n.label)	If m is the DTD node.
	removeDTDEl(n.label, m.label)	If n is an element.
	destroyDTDAtt(n.label, m.label)	If n is an attribute.
insertEdge (m, n,l, q)	insertDTDEl(n.label, ρ(e), m.label, q, d.label)	If n is an element
	addDTDAtt(n.label, m.label, t.label, d.label, v.label)	If node n represents an attribute.
insertEdgeAt (m, n, l, q, pos)	insertDTDEl(n.label, pos, m.label, q, d.label)	If n is an element
	addDTDAtt(n.label, m.label, t.label, d.label, v.label)	If node n represents an attribute.
deleteEdge (m, l)	removeDTDEl(n.label, m.label)	If n is an element.
	destroyDTDAtt(n.label, m.label)	If n is an attribute.
rename (n, l')	renameDTDEl(n.label, l')	if n is an element.

Table 14.10: Translation of Sangam graph Structural Change Primitives to XML Change Primitives.

between the object o_1 and o_3. Insert this object into R (e_2).

Similarly, the `delete-tuple` operation is translated to a sequence of deleteEdgeObject and deleteObject SAG-DU operations; and a `modify--tuple` operation into a renameValue operation. No edgeObject modifications are needed in this case as the OIDs of the objects do not change, just the values.

14.4 Reversing the Process

So far we have considered how local schema and data changes, i.e., changes in a relational schema or an XML DTD, can be translated into a sequence of Sangam graph operations. By necessity this translation must be reversable, i.e., we must be able to translate Sangam graph operations into a relational or XML operations. This reverse translation is simpler as it now deals with a simpler, perhaps more primitive set of changes. Recall from Chapter 11, that we use the *leaf* criteria to distinguish between an element and an attribute in the XML model and between a table and an attribute in the relational model. Thus, a *leaf* Sangam graph node (a node with no children) is an XML subelement if it has an edge that points to a node that gives its definition (a backpointer edge). It is an XML attribute otherwise. Similarly, a *leaf* node is always a relational attribute, while a non-leaf node is a table in the relational model. Based on this distinction, in Tables 14.10 and 14.11 we give the translation of the SAG-SC operations to XML and relational changes respectively.

Similar to the translation of the SAG-SC operations, the SAG-DU oper-

SAG Primitive	Relational Primitive	Description
insertNode (ln, τ, m, l, q)	CREATE TABLE ln	If m represents the schema node and n is not a leaf.
	ALTER TABLE m.label ADD COLUMN ln	If n represents an attribute i.e, n is a leaf and m is not a schema node.
insertNodeAt (ln, τ, m, l, q, pos)	same as above	
deleteNode (m, n)	DROP TABLE n.label	If n represents a table and if m represents a schema node.
	ALTER TABLE m.label DROP COLUMN n.label	If n represents an attribute and m a table.
insertEdge (m, n,l, q)	ALTER TABLE p.label ADD CONSTRAINT l FOREIGN KEY (m.label) REFERENCES n.label	If m and n both represent attributes. Here p is the parent node of the node m.
	CREATE TABLE n.label	If m represents the schema node and n is not a leaf.
	ALTER TABLE m.label ADD COLUMN n.label	If n represents an attribute i.e, n is a leaf and m represents a table.
insertEdgeAt (m, n,l, q, pos)	same as above	
deleteEdge (m, l)	ALTER TABLE p.label DROP CONSTRAINT l	If m represents an attribute, and e.toNode also represents an attribute. Here p represents the table that contains the node m, and e is the outgoing edge from m with label l.
	DROP TABLE p.label	If m represents a table. Here p = e.toNode and e is the edge with lable l outgoing from m and m represents a schema node.
	ALTER TABLE m.label DROP COLUMN n.label	If m represents a table. Here n = e.toNode() where e is the edge with label l
rename (n, l′)	ALTER TABLE p ALTER COLUMN n.label l′	if n represents an attribute. Here p is the parent node of the node n.

Table 14.11: Translation of Sangam graph Structural Change Primitives to Relational Change Primitives.

ations can also be translated into a set of data update operations in XML or the relational context. Tables 14.12 and 14.13 present the translation of the SAG-DU operations into a XML and relational data update operations respectively.

14.5 Summary

To summarize, in this chapter we have presented a set of basic update and evolution primitives for the Sangam graph model. We have shown that this set of operators are complete and have also shown that they are a sufficient set with respect to the evolution primitives provided for the relational and

SAG DU Primitive	XML DU Primitive	Description
addObject (v, n)	insert(v)	inserts the value v into the real-world representation of n
deleteObject (o, n)	delete(o.value())	Delete the value o.value from the entity represented by n.
rename (o, v′, n)	rename ((o.value()), v′)	Rename the value o.value to v′ of the entity represented by n.
addEdgeObject (o_1, o, e)	insert(content)	Insert the value o.value into the entity represented by e.toNode such that its parent value is o_1.value.
deleteEdgeObject (eo, e)	delete(eo.toObject.value)	Delete the value eo.toObject.value from the entity represented by n.toNode such that its parent value is eo.fromObject.value.

Table 14.12: Translation of Sangam graph Data Change Primitives to XML Data Change Primitives.

SAG DU Primitive	Relational DU Primitive	Description
addObject (v, n)	add-tuple v	inserts the value v into the real-world representation of n
deleteObject (o, n)	delete-tuple (o.value)	Delete the value o.value from the entity represented by n.
rename (o, v′, n)	modify-tuple((o.value()), v′)	Rename the value o.value to v′ of the entity represented by n.
addEdgeObject (o_1, o, e)	add-tuple (o_1.value, o.value)	Insert the value o.value into the entity represented by e.toNode such that its parent value is o_1.value.
deleteEdgeObject (eo, e)	delete-tuple (eo.fromObject.value, eo.toObject.value)	Delete the value eo.fromObject.value from the entity represented by e.toNode such that its parent value is eo.fromObject.value.

Table 14.13: Translation of Sangam graph Data Change Primitives to Relational Data Change Primitives.

XML models. That is, we have shown that all update and evolution primitives defined for the relational and XML models can be translated into a sequence of Sangam graph primitives, to produce the a change equivalent to the local change on the Sangam graph.

Chapter 15

Update Propagation

In Chapter 14, we present a taxonomy of Sangam graph primitives using that can be used to translate local schema and data changes into changes on the Sangam graph . However when a Sangam graph is mapped to another Sangam graph via a cross algebra graph, then to maintain consistency between the input Sangam graph and the derived output Sangam graph changes in the input Sangam graph must be reflected in the output Sangam graph . There are two possibilities for reflecting the change on the output Sangam graph . The first, *direct propagation*, reflects any change in the input on the output as well. The second, *transparent propagation*, hides when possible, from the output any change that is made on the input. While both mechanisms are achievable, in this dissertation we focus only on direct propagation.

In this chapter, we present the direct update propagation strategy for the cross algebra graph. For this, we first describe the overall propagation strategy for the cross algebra graph.

15.1 The CAT Propagation Strategy

15.1.1 Introduction

As presented in Chapter 12, the cross algebra graph consists of either context dependency or derivation trees or a composition of context dependency and derivation trees. Thus CAGs can be composed as a forest of one or more context dependency and derivation trees. This follows from the definition of the CAG (Definition 19). The propagation of a change through a CAG can thus be achieved by the propagation of this change through each individual tree (context dependency or derivation tree). In the forthcoming discussion, we focus only on the propagation strategy for the CAT. The only caveat when considering the CAT propagation is the propagation semantics for a change through a shared operator (Definition 16). We address this caveat as a separate point beyond the general propagation algorithms.

15.1.2 Updates on CAT

We now define a valid update on a Sangam graph .

Definition 24 (Valid Update) *Let SC denote the set of all operations on a Sangam graph as defined in Section 14. An operation* `c` $\in$ *SC on a node* `n` *or an edge* `e` *is* **defined** *for the SAG G if* `n` *(or* `e`*) either exists and/or is reachable by the specified path (for delete operations) or does not exist and/or is not reachable by the specified path (for add operations).*

As shown in Section 14.3, a local schema change, i.e., a change on the XML DTD or a relational schema, often results in a sequence of changes on

the SAG. To meet our primary goal of propagating a local schema change from the input Sangam graph to the output Sangam graph and hence eventually to the output application schema, we must thus consider the propagation of a sequence of changes through the cross algebra tree. We define a valid update sequence as follows.

Definition 25 (Valid Update Sequence) *A sequence* $\mathtt{u}_1, \mathtt{u}_2, \ldots \mathtt{u}_i \ldots \mathtt{u}_n$ *(*$1 < \mathtt{i} \leq \mathtt{n}$*) with* $\mathtt{u}_i \in SC$ *, denoted by* δG *, is valid for Sangam graph G if* $\mathtt{u}_i$ *is defined for the Sangam graph* G^{i-1} *that was obtained by applying* $\mathtt{u}_1, \mathtt{u}_2, \ldots \mathtt{u}_{i-1}$ *to G .*

15.1.3 Overall Propagation Strategy

For SQL views, many incremental update strategies have been proposed [Kel82, GB95]. Their essential focus is the calculation of the extent difference between the old `V` and new view `V'` by adding or subtracting tuples from the extent of the view `V`. In these works, the schemas of `V` and `V'` were assumed to stay the same. In the scenario when schema changes are considered [KR02] as in update propagation through SchemaSQL views by Koeller et al., the changes are made in-place on the individual algebra operators in the algebra tree. That is, the structure of the algebra tree remains the same and no algebra nodes are added or deleted.

However, when propagating schema changes through a CAT not only is the old output Sangam graph G different from the modified output Sangam graph G' , but also the structure of the CAT itself may be affected. New algebra nodes may be added as a result of an insertion and/or existing algebra nodes may be removed as a result of a deletion. We find that although

Function	Description
op.*toBeDeleted*()	Returns true if one of the inputs of operator ophave become invalid
op.*markForDeletion*()	Returns true and marks the operator opfor deletion in the cleanup pass of *Gen_Propagation*
op.*generateUpdate* (u)	Generates and returns an update u' based on the input update u .
op.*isAffectedBy*(u)	Returns true if the operator opis affected by u)
op.*contexts*()	Returns the number of parent operations for operator op, such that e:<op_p, op> == *context dependency*
op.*derivations*()	Returns the number of parent operations for operator op, such that e:<op_p, op> == *derivation*
op.*removeEdge*(e)	Removes the specified edge e outgoing from op
op.*children*()	Returns the number of children operators of the operator op. This includes all operators op_i that opderives from, and all op_i with which opis in context dependency relationship.
up_i.*getUpdate*()	Returns u , the update component of the update pair up_i
up_i.*getOperator*()	Returns op, the operator that generated the update u_i in the update pair up_i

Table 15.1: Brief Description of Functions Used in Algorithms of Figures 15.1 and 15.5.

data modification and deletion operations can be propagated through the CAT using the same propagation algorithm, we are unable to handle the insertion schema operations with the same. The main reason for this is the fact that even a simple insertNode operation can dramatically alter the CAG, and result in the construction of a new context dependency tree that must then be evaluated. Hence, we present two propagation algorithms: the *Gen_Propagation* algorithm to handle data updates and deletion operations, and the *Insert_Propagation* algorithm to handle the insertion schema operations.

The *Gen_Propagation* Algorithm

```
function List Gen_Propagation (Update u_p , Operator op_p)
{      List inSequence ← ∅,
   List δ ← ∅, //Used to gather the updates from children
   List δ' ← ∅ // Represents the updates sequence of the parent
   UpdatePair up , up_i
   Boolean updateApplied
   if (op_p    is leaf)
              up ← applyUpdate (u_p , op_p)
              δ.append (up )
   else
         //If not a leaf, then recursively invoke Gen_Propagation for all children
         for (all children op_i of op_p)
                      δ.append(Gen_Propagation (u_p , op_i))
         // δ denotes all updates generated by all children of op_p
         // Calculate the affect of each update u_i ∈ δ on op
         // Generate the parent's update sequence δ'
         for (all updatePairs up_i∈ δ) {
              u_i ← up_i.getUpdate()
                   up_i' ← applyUpdate (u_i ,op_p)
                   //Decide if up_i and up_i' must be
                   //appended to parent's sequence δ' of if up_i must be discarded
                   op_i ← up_i.getOperator()
                   if ((e:<op,op_i) == contextDependency)
                        // Calculate effect of the original update
                        // Check if update has been applied before
                        if (!updateApplied)
                             up ← applyUpdate (u_p , op_p)
                             updateApplied ← true
                        δ'.append(up_i)
                        δ'.append(up )
                        δ'.append(up_i' )
                   elseif ((e:<op, op_i>) == derivation)
                        δ'.append(up_i' )
         }
   return δ';
}
```

Figure 15.1: First Pass of Cross Algebra Tree Target Maintenance Algorithm.

The First Pass: Propagating the Update. The *Gen_Propagation* algorithm works as a two-step process. The first pass of the algorithm performs a post-order traversal of the algebra tree. This ensures that each operator processes the input updates after all its children have already computed their output. Since the cross algebra tree is connected and cycle-free (Lemma 7), all operators will be visited exactly once. If an operator is shared, then it will contribute an update, if applicable, to all its parent operators. If the shared operator contributes directly to the final output Sangam graph G', then the evaluation semantics for shared operators (Definition 16) are applicable for the propagation of change, i.e., only one update will be recorded in the final update sequence.

We can assume here without any loss of generality that each tree has its copy of each shared operator. After all operators have been visited, the output of the algorithm will be an update sequence δ to be applied to the output Sangam graph G' after the completion of the second pass.

```
function UpdatePair applyUpdate (Update u_p , Operator op_p)
{       if (op_p.isAffectedBy(u_p ))
         u_p' ← op_p.generateUpdate (u_p )
         if (op_p.toBeDeleted())
               op_p.markForDeletion()
         up ← (u_p' ,op_p)
   return up
}
```

Figure 15.2: First Pass of Cross Algebra Tree Target Maintenance Algorithm - The UpdatePair Function.

Figure 15.1 depicts the first pass of the *Gen_Propagation* algorithm. Table 15.1 lists the functions and their brief descriptions as used in Figure 15.1.

Here, each node in the algebra tree is aware of the algebra operator it represents (see Section 7 for algebra operators). At any given time, an operator op_i can accept *one* input update u_i, where $u_i \in$ SC , and will generate as output *one* update u_i' to be applied on the output Sangam graph G' . The output update for each operator is recorded as a pair $<u_i', op_i>$ in a local *update sequence* δ, where u_i' is the output update and op_i is the operator to which the input update u_i was applied to produce u_i'. The updates u_i and u_i' are Sangam graph operation (Section 14) and are specified with all their parameters. We assume that given an update operation, an algebra operator can detect whether or not it is affected by the change. If it is not affected, it will simply return the input update u_i as its output.

After all the updates u_i for the children of an operator op_p are computed and collected in a sequence (denoted by variable δ in the algorithm), each update $u_i \in \delta$ is propagated through op_p, irrespective of if the child op_i is related to op_p via derivation; or if child op_i is related to op_p via context dependency. In either case, a new update u_i' will be produced by the parent operator op_p if the operator op_p is effected by the update u_i produced by the child operator op_i. This update u_i' is recorded as an update pair (u_i', op_p) and is appended to the update sequence δ' produced by the operator op_p.

The edge, context dependency or derivation, between the parent and the child operator affects the parent operator in two ways: (1) in deciding whether the original update u must be applied to the parent operator op_p; and (2) in deciding whether the update sequence generated by the child operator must be propagated up, i.e., included in the update sequence gen-

erated by the parent operator.

In general, a parent operator $\texttt{op}_p$ is not affected by the original update $\texttt{u}$ if none of its children are affected by the same update $\texttt{u}$ and *all* the children are related to the parent operator by derivation. If however, the parent is related to a child operator by a context dependency edge, then the parent may be affected by the update $\texttt{u}$ even if the child operator is not affected. Hence, if a parent has a context dependency edge to its child operator, then the original update $\texttt{u}$ must be applied to the parent operator $\texttt{op}_p$. If the operator $\texttt{op}_p$ is affected by the update $\texttt{u}$, then a new update $\texttt{u}'$ and a corresponding update pair $\texttt{up}'$ are generated. The update pair $\texttt{up}'$ is appended to the update sequence δ' produced by $\texttt{op}_p$. The original update $\texttt{u}$ is applied only once. The original update is not appended into the output update sequence δ' generated by an operator, but is stored as a ***global update***. This propagation of the update $\texttt{u}$ is indicated by the following lines in the *Gen_Propagation* algorithm given in Figure 15.1:

// *Calculate effect of the original update*

$\texttt{up} \leftarrow$ ***applyUpdate*** ($\texttt{u}_p$, $\texttt{op}_\texttt{p}$)

We next consider the effect of the type of edge that exists between $\texttt{op}_p$ and $\texttt{op}_i$ on the generation of the update sequence δ' produced by the parent operator $\texttt{op}_p$. Recall from Chapter 12 that in a derivation tree only the output of the root operator is visible in the final output produced by the tree. It therefore follows that if there exists a derivation relationship between $\texttt{op}_p$ and $\texttt{op}_i$, only the update $\texttt{u}_i'$ generated by $\texttt{op}_p$ will appear in the update sequence δ' generated by $\texttt{op}_p$. Hence, after producing the update $\texttt{u}_i$',

the update pair (u_i, op_i) is discarded, and the update pair (u_i', op_p) is appended to the update sequence δ'[1] generated by the parent operator op_p. This is given by: δ'.***append***(up_i') in the ***Gen_Propagation*** algorithm depicted in Figure 15.1. On the other hand, in a context dependency tree the outputs of all children operators and the root operator appear in the final output produced by the tree. Thus, if there exists a context dependency edge between the parent operator op_p and the child operator op_i, then the update pair (u_i, op_i) is appended to the update sequence δ' produced by the parent operator op_p. To maintain the dependency between the parent and the child operator, the update pair (u_i', op_p) is appended after the update pair (u_i, op_i) in the update sequence δ'. This is given by the pair of statements: δ'.***append***(up_i'), δ'.***append***(up_i), in the ***Gen_Propagation*** algorithm depicted in Figure 15.1.

For any operator op_i if the update u_i is such that it removes the input of the operator op_i, then the operator op_i is marked for deletion. The actual removal of the operator from the CAG occurs in the second pass, ***Gen_CleanUp*** , of the ***Gen_Propagation*** algorithm.

The Second Pass: Cleaning Up. The purpose of the second pass is to clean up the cross algebra tree and to ensure that the output Sangam graph G' after the application of δ', the update sequence produced by pass 1, and the clean-up still produces a valid Sangam graph . In this clean-up process, we make the decision that:

1. if a parent operator op_p that has a derivation edge to a child operator

[1] Initially the update sequence δ' is empty.

$\mathtt{op}_i$ is removed, then if the child operator $\mathtt{op}_i$ is not a shared operator, then $\mathtt{op}_i$ is also removed. This follows from the fact that (1) the output of a child operator is not directly visible in the output produced by the parent operator; and (2) we do not allow partial derivation results to be reflected in the output Sangam graph G'. As an example, consider the derivation tree DT depicted in Figure 15.3 and given by the expression:

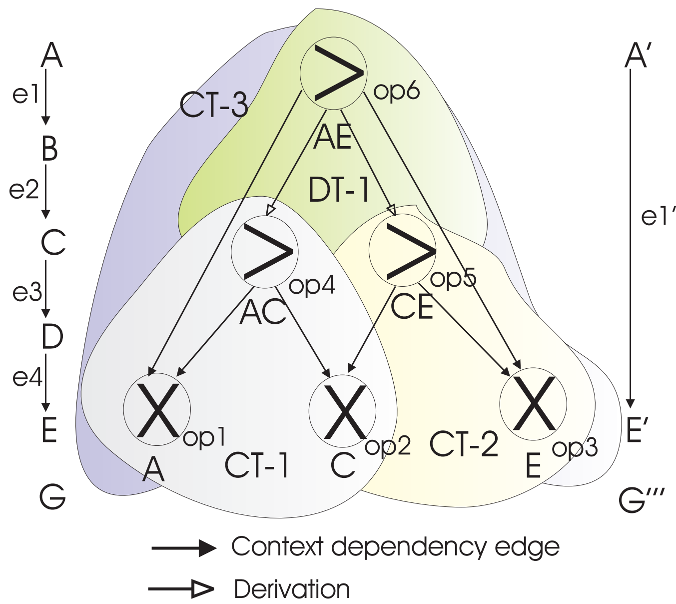


Figure 15.3: A Cross Algebra Graph.

$$\mathtt{DT} = (\mathtt{op6}_{\mathtt{e'}:<\mathtt{A'},\mathtt{E'}>}(\mathtt{CT1},\mathtt{CT2})) \qquad (15.1)$$

where CT1 and CT2 are given as:

$$\texttt{CT1} = (\texttt{op4}_{\texttt{eTemp1}:<\texttt{A}',\texttt{C}'>}(\texttt{e1} :< \texttt{A}, \texttt{B} >, \texttt{e2} :< \texttt{B}, \texttt{C} >), (\texttt{op1}_{\texttt{A}'}(\texttt{A}) \circ \texttt{op2}_{\texttt{C}'}(\texttt{C}))) \tag{15.2}$$

$$\texttt{CT2} = (\texttt{op5}_{\texttt{eTemp2}:<\texttt{C}',\texttt{E}'>}(\texttt{e3} :< \texttt{C}, \texttt{D} >, \texttt{e4} :< \texttt{D}, \texttt{E} >), (\texttt{op2}_{\texttt{C}'}(\texttt{C}) \circ \texttt{op3}_{\texttt{E}'}(\texttt{E}))). \tag{15.3}$$

Here the output produced by DT is the output of the root operator op6, e':<A', E'>. The outputs of the context dependency graphs CT1 and CT2 do not appear in the final output Sangam graph . Hence, in general it follows that if the root operator $\texttt{op}_r$ of a derivation tree DT is removed, then all its non-shared children operators $\texttt{op}_i$ (including the operators that are part of the the context dependency trees CT1 and CT2) must also be removed as their output is not visible in the final output G'. Figure 15.4 depicts the CAG after this step.

2. if a parent operator $\texttt{op}_p$ is removed such that there exists a context dependency edge between the parent $\texttt{op}_p$ and the child $\texttt{op}_i$, then the child operator $\texttt{op}_i$ is not affected. This follows from the fact that in a context dependency tree CT the outputs of all operators appear in the final output of the tree. In the example CAG given in Figure 15.3, the updated context dependency tree CT3 will be as depicted in Figure 15.4.

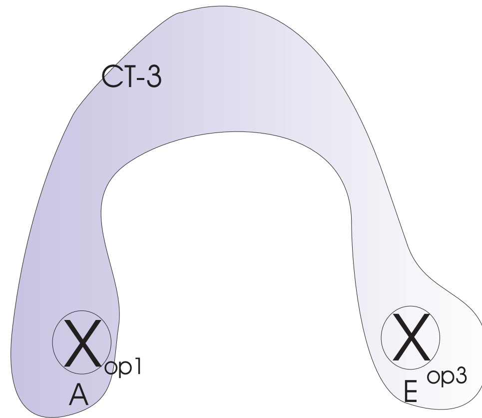


Figure 15.4: The Updated Cross Algebra Graph After Step 1.

Figure 15.5 gives the second pass of the *Gen_Propagation* algorithm called *Gen_CleanUp* . The algebra tree is traversed in in-order as the deletion of a parent operator may affect the child operator. In the second pass *Gen_CleanUp* , for each operator $\mathtt{op}_p$ marked for deletion in the first pass *Gen_Propagation* , if the operator $\mathtt{op}_p$ has children operators $\mathtt{op}_i$ connected via derivation, then all non-shared operators $\mathtt{op}_i$ are also marked for deletion. If the operator $\mathtt{op}_i$ is shared, then the derivation edge between $\mathtt{op}_p$ and $\mathtt{op}_i$ is marked for deletion. If the operator $\mathtt{op}_p$ has children operators $\mathtt{op}_i$ connected via context dependency, then the children operators $\mathtt{op}_i$ remain unaffected. Once all children of an operator are visited and marked for deletion where appropriate, all children (and edges) marked for deletion are removed as well as the parent (if marked for deletion). The final update sequence δ' is then applied to the output Sangam graph `G'`.

```
function Gen_CleanUp (Operator op)

  if (op.markForDeletion () )
      // Propagate the markForDeletion to all derivation children
      for (all children op_i of op)
          if ((e:<op, op_i) == derivation)
          if (op_i is not shared)
                  // Mark the child operator for deletion.
                  op_i.markForDeletion()
              else
                  // Mark the derivation edge for deletion.
                  e.markForDeletion()

  //Recursivly invoke the algorithm
  for (all children op_i of op)
      Gen_CleanUp (op_i)

  // Remove the operator
  if (op.markForDeletion () )
      delete (op)

  return
```

Figure 15.5: Second Pass - Clean Up of Cross Algebra Tree Target Maintenance Algorithm.

Propagation through a CAG. If a CAG is composed of one or more CAT, i.e., $\texttt{CAG} = \texttt{CAT}_1 \circ \texttt{CAT}_2 \circ \ldots \texttt{CAT}_n$, then the *Gen_Propagation* algorithm is applied to each $\texttt{CAT}_i$, for $\texttt{i} = \texttt{1}$ to $\texttt{n}$. If the resultant updated output Sangam graph `G''` is not a well-formed Sangam graph as per Definition 21, then the entire update process is aborted and the effects of the application of the sequence of updates δ' on the output Sangam graph `G'` are rolled back.

Propagation through Shared Operators. Recall from Chapter 12 that an operator can be shared by two different CATs but it is evaluated only once

and hence its output directly appears at most one time in the output Sangam graph G'. If however a shared operator is affected by a change then the change must be propagated through it as well as all the parent operators to which it is linked either via a derivation or a context dependency edge. The update produced by the shared operator appears only once in the final update sequence δ'. This can be easily ensured by checking all the update pairs in the final update sequence δ'.

The *Insert_Propagation* Algorithm

The propagation of the insertNode , insertNodeAt , insertEdge , and insertEdgeAt operations through the CAT to the output Sangam graph G' is treated differently than the propagation of all other schema and data update operations. This is mostly due to the nature of these operations and their mapping via the cross algebra operators to the output Sangam graph .

The insertNode and insertNodeAt Operations. Consider the CAG depicted in Figure 15.6 and given by the expression:

$$\mathtt{CAG} = (\mathtt{CT1} \circ \mathtt{CT2}) \tag{15.4}$$

where $\mathtt{CT1} = \mathtt{op3}_{(e1')}(\mathtt{e1}), (\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op2}_{\mathtt{B'}}(\mathtt{B}))$

$\mathtt{CT2} = \mathtt{op5}_{(e2')}(\mathtt{e2}, \mathtt{e5}), (\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op4}_{\mathtt{C'}}(\mathtt{C}))$

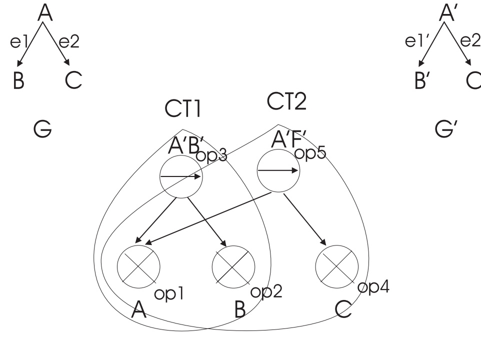


Figure 15.6: An Example CAG.

Now consider that the operation insertNode (D, τ, A, e1, 1:1), where D is the label of the new node that is to be inserted into the Sangam graph G, and e1 is the label of the edge e:<A, D>[2]. The resultant Sangam graph G_u, after the application of the insertNode operation, is shown in Figure 15.7. The insertNode operation creates a node as well as an edge. Given that we employ the semantics that all changes in the input must be visible in the output, to propagate this insertNode operation to the derived output Sangam graph G', we must now map the newly created node D and the corresponding edge e:<A,D> to the output Sangam graph G'. For this we would need to:

1. Given a CAG, find the cross algebra operator op_1 that maps the node A to produce the output A' $\in$ G'. If no such operator exists in the CAG, then a new cross operator is created such that it maps A to A' $\in$ G';

[2]To make it easier to read, we use the labels of nodes here to refer to the actual node objects.

2. create a new cross operator $\texttt{op}_6$ that maps the node D to D′ ∈ G′; and

3. create a connect ⊖ operator $\texttt{op}_7$ that ensures that the objects of A′ will provide the context for the objects of D′.

In effect, we create a new CAT CT3 given by the expression $\texttt{CT3} = \texttt{op7}_{(e3')}(\texttt{e3}), (\texttt{op1}_{\texttt{A}'}(\texttt{A}) \circ \texttt{op6}_{\texttt{D}'}(\texttt{D}))$, and as depicted in Figure 15.8.

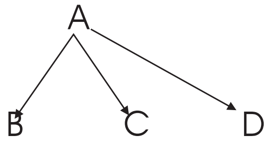


Figure 15.7: Example: Modified Input Sangam graph $\texttt{G}_u$ produced by Application of insertNode (D, τ, A, e1, 1:1) on the input Sangam graph G in Figure 15.6.

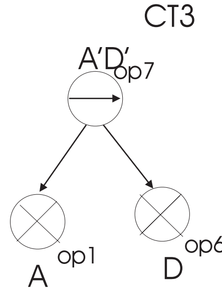


Figure 15.8: Example: The New CAT CT3 Produced by insertNode and insertNodeAt Propagation Steps.

The final updated CAG CAG′ is as depicted in Figure 15.9 and is given by the expression:

$$\texttt{CAG}' = (\texttt{CT1} \circ \texttt{CT2} \circ \texttt{CT3}) \tag{15.5}$$

where $\texttt{CT1} = \texttt{op3}_{(e1')}(\texttt{e1}), (\texttt{op1}_{\texttt{A}'}(\texttt{A}) \circ \texttt{op2}_{\texttt{B}'}(\texttt{B}))$

$\texttt{CT2} = \texttt{op5}_{(e2')}(\texttt{e2}, \texttt{e5}), (\texttt{op1}_{\texttt{A}'}(\texttt{A}) \circ \texttt{op4}_{\texttt{C}'}(\texttt{C}))$

$\texttt{CT3} = \texttt{op7}_{(e3')}(\texttt{e3}), (\texttt{op1}_{\texttt{A}'}(\texttt{A}) \circ \texttt{op6}_{\texttt{D}'}(\texttt{D}))$

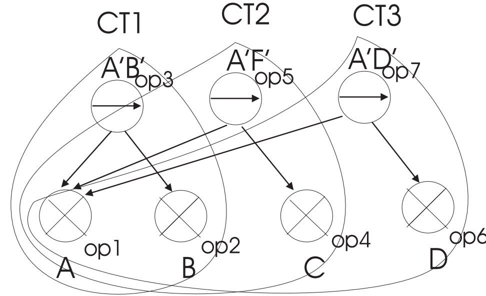


Figure 15.9: Example: Modified CAG `CAG'` after the Addition of the new CAT `CT3`.

These steps are in general the default steps taken to handle the insertNode and insertNodeAt operations. To propagate the update to the output Sangam graph `G'`, we produce an update sequence δ that corresponds to the evaluation of the evaluation of the newly constructed CAT `CT3` (refer Section 12.3.1). Thus, if `out` denotes the output of `CAG`, $\texttt{out}_3$ the output of `CT3`, and `out'` the output of `CAG`', then $\texttt{out'} = \texttt{out} \bigcup \texttt{out}_3$.

The new CAT `CT3` and hence also the updated CAG `CAG'` are well-formed CAG (Definition 21). Thus, the evaluation of `CAG'` will always produce a valid Sangam graph $\texttt{G}_u\texttt{'}$, where $\texttt{G}_u\texttt{'}$ is the updated output Sangam graph .

The insertEdge and insertEdgeAt Operations. Next consider the insertEdge and insertEdgeAt operations. These operations insert a new edge between two given nodes. Consider the input Sangam graph given in Figure 15.10. Now consider that the operation insertEdge (`A`, `E`, `l`, `q`) creates a new edge `e`:<`A`, `E`> between the nodes `A` and `E` to result in the modified Sangam graph $\texttt{G}_u$ shown in Figure 15.11. To enable the mapping of this ad-

dition to the output Sangam graph we do the following sequence of steps:

Figure 15.10: Example of Input Sangam-graph G.

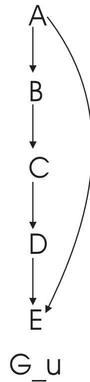


Figure 15.11: Example of Modified Input Sangam graph G_u.

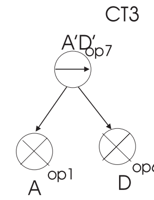


Figure 15.12: Example of The New CAT Produced by insertEdge and insertEdgeAt Propagation Steps.

1. Given a CAG, find the cross operators that map the node A to A' $\in$ G', and the node E to E' $\in$ G', where G' is the output Sangam graph of the CAG. If no such operators are found, then create two cross operators op_1 and op_2 to map the two nodes A and E respectively.

2. create a connect operator op_3 that ensures that the objects of A' will provide the context for the objects of E'. The new CAT, CT $\in$ CAG is shown in Figure 15.12.

These are the general steps taken to handle the insertEdge and the insertEdgeAt operations. The new CAT CT (Figure 15.12) can be evaluated to get the modified output Sangam graph G''. Let CAG represent the CAG prior to the update such that its evaluation produces the output Sangam

graph G'. After the propagation of the insertEdge operation, the modified CAG CAG' = CAG $\circ$ CT. The output produced by CAG' is out' = out $\bigcup$ $\texttt{out}_C$, where out is the output of CAG and $\texttt{out}_C$ the output produced by the CAT CT. We only need to evaluate CT here to update G' and produce the desired output Sangam graph G''.

The new CAT CT and hence also the updated CAG CAG' are well-formed CAGs (Definition 21). Thus, the evaluation of CAG' will always produce a valid Sangam graph $\texttt{G}_u$', where $\texttt{G}_u$' is the updated output Sangam graph .

15.2 Default Propagation Through the Cross Algebra Operators

15.2.1 Propagation Rules for SAG-SC Operations

In Tables 15.2-15.5 we give the default propagation tables for the four cross algebra operators. Table 12.1 presents the notation used in these tables. Each update case in Tables 15.2-15.5 indicates the update operation generated by the cross algebra operator when an input update operation is applied to it as well as any change to the operator itself. An operator is affected by a change when the conditions specified are true, and not when the conditions are false. The insertion operators are not considered here as they have special default cases as given by the *Insert_Propagation* algorithm.

Inspection of the propagation update tables will show that in some cases the algebra operator is not affected by the change. This is a result

of the granularity of the algebra operator. Each algebra operator takes as input at most two nodes or edges. If the update does not directly or indirectly (via derivation) affect the input nodes or edges of the algebra node, then the algebra operator is not affected by the change during update propagation.

SAG Operation	Condition	Update Produced	Algebra Update
deleteNode (m, n)	**if** n′ does not have any children nodes && $\otimes_{m'}$(m) and $\otimes_{n'}$(n) exist && edge $\ominus_{e'}$ (e) exists && $\otimes_{n'}$(n) is not shared	deleteNode (m′, n′)	mark $\otimes_{n'}$(n) for deletion
	else	no effect	no effect
deleteEdge (m, l)	n/a	n/a	n/a
rename (n, l′)	if $\otimes_{n'}$(n) exists	rename (n′, l′)	no effect
	else	no effect	no effect

Table 15.2: Default Propagation Rules for $\otimes_{n'}$ (n).

Propagation By Example: A SAG-SC Update Propagation Example

Example 7 *Consider the cross algebra graph in Figure 15.13. Here the CAG operates on input Sangam graph* G *and produces as output the Sangam graph* G′*. The edges of* G *are* e1:<A,B>, e2:<A,C>, e3:<B,D>, e4:<B,E> *and* e5:<C,F>*. Similarly for* G′*, the edges are* e1′:<A′,B′>, e2′:<A′,F′>, e3′:<B′,D′>, *and* e4′:<B′,E′>*. The expression for this is given as:*

SAG Operation	Condition	Update Produced	Algebra Update
deleteNode (m, n)	**if** e′:<m′,n′> exists && n′ has no children	deleteEdge (m′, l) where l is label of edge e′	mark $\ominus_{e'}$ (e) for deletion
	else	no effect	no effect
deleteEdge (m, l)	**if** e′:<m′,n′> exists with label l && n′ has $>$ 1 incoming edges	deleteEdge (m′,l)	no effect
	if e′:<m′,n′> exists with label l && n′ has $=$ 1 incoming edges	deleteEdge (m′,l)	mark $\ominus_{e'}$ (e) for deletion
	else	no effect	no effect
rename (e, l′)	**if** e′:<m′,n′> exists with label l	rename (e′,l′)	no effect

Table 15.3: Default Propagation Rules for $\ominus_{e'}$ (e) where edge e:<m,n> and edge e′:<m′,n′>.

$$\mathtt{CAG} = (\mathtt{CT1} \circ \mathtt{CT2} \circ \mathtt{CT3} \circ \mathtt{CT4}) \tag{15.6}$$

where $\mathtt{CT1} = \mathtt{op3}_{(e1')}(\mathtt{e1})$, $(\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op2}_{\mathtt{B'}}(\mathtt{B}))$

$\mathtt{CT2} = (\mathtt{op5}_{(e2')}(\mathtt{e2}, \mathtt{e5})$, $(\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op4}_{\mathtt{F'}}(\mathtt{F})))$

$\mathtt{CT3} = \mathtt{op7}_{(e3')}(\mathtt{e3})$, $(\mathtt{op2}_{\mathtt{B'}}(\mathtt{B}) \circ \mathtt{op6}_{\mathtt{D'}}(\mathtt{D}))$ *and*

$\mathtt{CT4} = \mathtt{op9}_{(e4')}(\mathtt{e4})$, $(\mathtt{op2}_{\mathtt{B'}}(\mathtt{B}) \circ \mathtt{op8}_{\mathtt{E'}}(\mathtt{E}))$.

Now consider that the input Sangam graph G *is modified and the node* E *is removed via the Sangam graph operation: deleteNode (*B, E*). Recall from Chapter 14 that the deletion of a node also results in the deletion of the edge* e4:<B,E>.

SAG Operation	Condition	Update Produced	Algebra Update
deleteNode (m, p)	**if** $\ominus_{e1'}$(e$_1$, e$_2$) exists such that e$_1$:<m,n> and e$_2$:<n, o> with p = n or o, and e1′:<m′,o′> && o′ has > 1 incoming edges	deleteEdge (m′, l′) where l′ is the label of edge e1′	mark $\ominus_{e1'}$(e$_1$, e$_2$) for deletion
	else	no effect	no effect
deleteEdge (m, l)	**if** $\ominus_{e1'}$(e$_1$, e$_2$) exists such that e$_1$:<m,n> and e$_2$:<n, o> with p = n or o, and e1′:<m′,o′> && o′ has > 1 incoming edges	deleteEdge (m′, l) where l is the label of the edge e1′	mark $\ominus_{e1'}$(e$_1$, e$_2$) for deletion
	else	no effect	no effect
rename (e$_1$, l′)	**if** $\ominus_{e1'}$(e$_1$, e$_2$) exists such that e$_1$:<m,n> and e$_2$:<n, o> with p = n or o, and e1′:<m′,o′>	rename (e1′,l′)	no effect

Table 15.4: Default Propagation Rules for $\ominus_{e1'}$(e$_1$, e$_2$).

Let u$_1$ = *deleteNode (*B, E*) denote the update applied to the CAG shown in Figure 15.13. The change is applied to each of the four CATs:* CT1, CT2, CT3 *and* CT4 ∈ CAG*. Let us first consider the affect of the update* u$_1$ *on the CAT* CT1*. None of the operators* op1, op2 *and* op3 ∈ CT1 *are affected by this update (Tables 15.2 and 15.3, Rules 1 and 2). Hence the CAT* CT1 *is not affected by the update. Similarly, the CATs* CT2 *and* CT3 *are not affected either.*

SAG Operation	Condition	Update Produced	Algebra Update
deleteNode (m, o)	**if** $\oslash_{e1',e2',n'}$(e) exists such that e:<m,o> and e1':<m',n'> and e2':<n',o'> && o' has > 1 incoming edges && n' has exactly 1 incoming edge and 1 outgoing edge	deleteNode (m', n'), deleteEdge (m', l1), deleteEdge (n', l2) where l1 and l2 are the labels of e1' and e2' respectively	mark $\oslash_{e1',e2',n'}$(e) for deletion
	else	no effect	no effect
deleteEdge (m, l)	**if** $\oslash_{e1',e2',n'}$(e) exists such that e:<m,o> and e1':<m',n'> and e2':<n',o'> && o' has > 1 incoming edges && n' has exactly 1 incoming edge and 1 outgoing edge	deleteEdge (m', l1), deleteNode (m', n'), deleteEdge (n', l2')	mark $\oslash_{e1',e2',n'}$(e) for deletion
	else	no effect	no effect
rename (e, l)	**if** $\oslash_{e1',e2',n'}$(e) exists such that e:<m,o> and e1':<m',n'> and e2':<n',o'>	rename (e1, l), rename (e2, l)	no effect

Table 15.5: Default Propagation Rules for $\oslash_{e1',e2',n'}$(e).

However, when the update is applied to the leaf operators op2 *and* op8 *of the CAT* CT4 *(as per the* Gen_Propagation *algorithm in Figure 15.1), we find that*

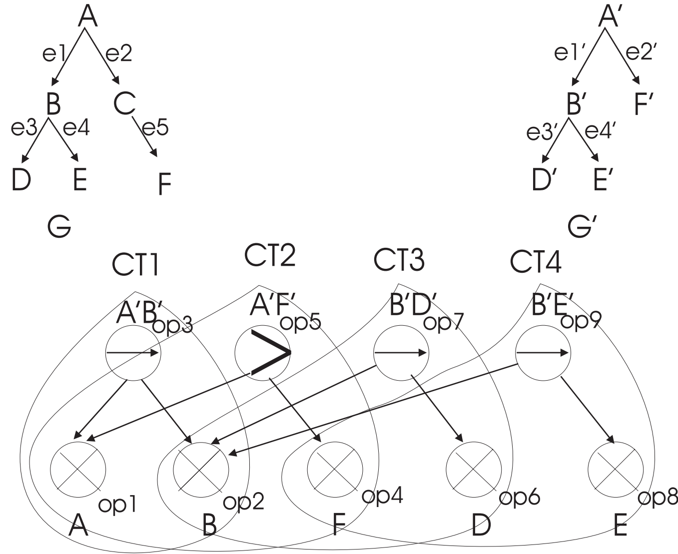


Figure 15.13: Another Example of a Cross Algebra Graph.

while the operator op2 *is not affected, the operator* op8 *is affected by the update* u_1*. Thus the CAT* CT4 *is affected by the update* u_1*. As the update does not affect the operator* op2 $\in$ CT4 *the update sequence generated by* op2 *will be an empty sequence. However, by Rule 1 in Table 15.2, when the original update* u_1 = *deleteNode (*B, E*) is applied to the operator* op8*, it will produce an update* u_1*'* = *deleteNode (*B', E'*). The update sequence produced by operator* op8 *is therefore*[3]*:*

$$\delta' =< u_1' > \tag{15.7}$$

Moreover, as the input of the operator op8 *is deleted (node* E*), the operator* op8 *is marked for deletion.*

[3]We list only the updates here, and not the complete update pairs.

By the Gen_Propagation *algorithm (Figure 15.1), this update sequence is propagated and applied to the parent operator(s)* `op9` *of* `op8` *along with the original update* u_1. *By Rule 1 in Table 15.3 it can be seen that the update* u_1 *affects the operator* `op9` *producing an update* u_2*' = deleteEdge (*`B', l'`*) where* `l'` *is the label of the edge* `e4'`*:<*`B', E'`*>. It is not affected by the update* u_1*' ∈ δ'. As per the propagation algorithm,* Gen_Propagation *, if two operators have a context relationship, then the output update sequence generated by the parent operator is a concatenation of the update sequence of the child operator and the updates produced by the parent operator. Hence, the update sequence δ" generated by the operator* `op9` *is given as:*

$$\delta'' =< u'_1, u'_2 > \tag{15.8}$$

As `op9` *is the root of the context tree* `CT4`*, the propagation terminates at it. The update sequence δ" generated by the operator* `op9` *is the final update sequence. Moreover, as one input of the operator* `op9` *is deleted, namely, the edge* `e4`*:<*`B, E`*>, the operator* `op9` *is marked for deletion.*

In pass 2, the Gen_CleanUp *algorithm (given in Figure 15.5), all operators,* `op9` *and* `op2` *in this case, marked for deletion are removed. The context dependency edge between the operators* `op9` *and* `op2` *is also removed as a result of the fact that at the end of the* Gen_CleanUp *algorithm* `op9` *no longer exists. The affect of these removals is the deletion of the CAT* `CT4`*. Thus, at the end of the pass 2 (*Gen_Propagation *2 algorithm) we have the CAG given by the following expression:*

$$\mathtt{CAG} = (\mathtt{CT1} \circ \mathtt{CT2} \circ \mathtt{CT3}) \tag{15.9}$$

where CT1, CT2 *and* CT3 *are as before. At the end of pass 2, the update sequence* δ*" is applied to the output Sangam graph* G' *to produce the update output Sangam graph* G''*. This updated output Sangam graph* G'' *does not include the node* E' *nor the edge* e4':<B', E'>*. In other words, the updated output Sangam graph* G'' *is exactly the output of the modified CAG* CAG' *given in Equation 15.9.*

Example 8 *As a second example, consider the cross algebra graph in Figure 15.14. The expression for this is given as:*

$$\mathtt{CT3} = (\mathtt{DT1}, (\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op3}_{\mathtt{E'}}(\mathtt{E}))) \tag{15.10}$$

where DT1 *is:*

$$\mathtt{DT1} = \begin{array}{c} (\mathtt{op6}_{\mathtt{e'}:<\mathtt{A'},\mathtt{E'}>}((\mathtt{op4}_{\mathtt{eTemp1}:<\mathtt{A'},\mathtt{C'}>}(\mathtt{e1}:<\mathtt{A},\mathtt{B}>, \mathtt{e2}:<\mathtt{B},\mathtt{C}>), \\ (\mathtt{op1}_{\mathtt{A'}}(\mathtt{A}) \circ \mathtt{op2}_{\mathtt{C'}}(\mathtt{C}))), \\ (\mathtt{op5}_{\mathtt{eTemp2}:<\mathtt{C'},\mathtt{E'}>}(\mathtt{e3}:<\mathtt{C},\mathtt{D}>, \mathtt{e4}:<\mathtt{D},\mathtt{E}>), \\ (\mathtt{op2}_{\mathtt{C'}}(\mathtt{C}) \circ \mathtt{op3}_{\mathtt{E'}}(\mathtt{E}))))) \end{array} \tag{15.11}$$

Here cross operator op1 *maps the input node* A *to the output node* A'*, the cross operator* op2 *maps the input node* C *to produce an intermediate output* C' *(not*

seen in the final output), and cross operator op3 *maps the node* E *to the output node* E'*. The operators* op4*,* op5 *and* op6 *are smooth operators used to flatten out the edges in the input to produce one edge* e:<A', E'> *in the output Sangam graph* G'*. The operator* op6 *has a context dependency on operators* op1 *and* op3 *and a derivation dependency on operators* op4 *and* op5*.*

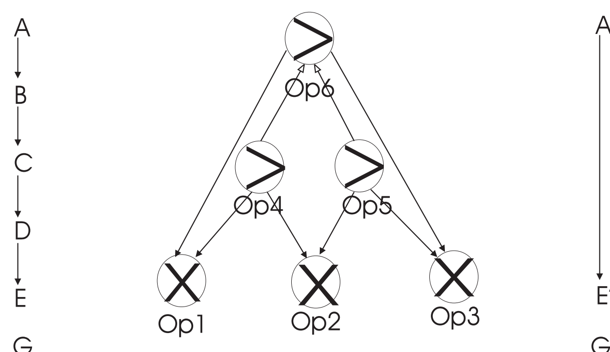


Figure 15.14: A CAT Example that maps the input Sangam graph G to the output Sangam graph G'.

Now consider that the input Sangam graph G *is modified and the node* E *is removed via the Sangam graph operation: deleteNode (*D, E*). Recall from Chapter 14 that the deletion of the node* E *also results in the deletion of the edge* e:<D, E>*. Let* u_1 = *deleteNode (*D, E*) denote the update applied to the CAT in Figure 15.14. Based on the update propagation algorithm given in Figure 15.1, we begin with applying the update to the leaves of the CAT* CT3*. Thus, the update is applied to* op_1, op_2 *and* op_3*. The update applied to* op_1 *and* op_2 *will result in empty update sequences as the two operators are both not affected by this update. The operator* op_3 *is affected by this update (Table 15.2, Rule 1) and will generate an update sequence:*

$$\delta = < \mathtt{u}_1' > \tag{15.12}$$

where $\mathtt{u}_1{}' = $ *deleteNode* ($\mathtt{D}'$, $\mathtt{E}'$). *Moreover, as* $\mathtt{E}$ *the input of this operator has been deleted, the operator* $\mathtt{op}_3$ *is marked for deletion (Table 15.2, Rule 1).*

The operator $\mathtt{op}_3$ *has two parents* $\mathtt{op}_5$ *and* $\mathtt{op}_6$. *Based on the* Gen_Propagation *algorithm (Figure 15.1), the update sequence* δ *(Equation 15.12) produced by the children operators is applied to the parent operator in order from left to right. Let us first consider the operator* $\mathtt{op}_5$. *The update* $\mathtt{u}_1{}' \in \delta$ *is applied to* $\mathtt{op}_5$. *The update* $\mathtt{u}_1$ *does not affect the operator* $\mathtt{op}_5$. *However, the original update* $\mathtt{u}_1$ *does affect the operator* $\mathtt{op}_5$ *(Table 15.4, Rule 1) and generates an update* $\mathtt{u}_2' =$ *deleteEdge* ($\mathtt{C}'$, $\mathtt{l}$) *where* $\mathtt{l}$ *is the label for edge* $\mathtt{e}'$:<$\mathtt{C}'$, $\mathtt{E}'$>. *As the operator* $\mathtt{op}_3$ *is in context relationship with the operator* $\mathtt{op}_5$, *the update sequence generated by the operator* $\mathtt{op}_5$ *is a concatenation of the update produced by* $\mathtt{op}_3$, $\mathtt{u}_1'$ *and the update produced by* $\mathtt{op}_5$, *i.e.* $\mathtt{u}_2'$. *Hence,*

$$\delta' = < \mathtt{u}_1', \mathtt{u}_2' > \tag{15.13}$$

where $\mathtt{u}_2'$ *and* $\mathtt{u}_1'$ *are as given above. Moreover, as one of the inputs of the operator* $\mathtt{op}_5$, *the edge* $\mathtt{e}$:<$\mathtt{D}$, $\mathtt{E}$> *is deleted, the operator* $\mathtt{op}_5$ *is marked for deletion (Table 15.4, Rule 1).*

Similarly, the update sequence $\delta = < \mathtt{u}_1' >$ *produced by* $\mathtt{op}_3$ *is also applied to the operator* $\mathtt{op}_6$. *Based on the update* $\mathtt{u}_1'$ *and* $\mathtt{u}_1$, *the condition given in Table 15.4*

Rule 1 is false. Hence the operator $\texttt{op}_6$ *is not affected by the update* $\texttt{u}_1$*' or the update* $\texttt{u}_1$*. However, as the operator* $\texttt{op}_6$ *has a context relationship with operator* $\texttt{op}_3$*, even though* $\texttt{op}_6$ *is not affected by the updates in* δ*, the update sequence* δ *is included in the final update sequence produced by operator* $\texttt{op}_6$*. Thus,*

$$\delta'' = \delta_{op3}{}^{4}. =< \texttt{u}_1{}' > \tag{15.14}$$

where δ*" is the update sequence produced by* $\texttt{op}_6$*.*

Now consider again the operator $\texttt{op}_6$ *that derives from the operator* $\texttt{op}_5$*, i.e.,* $\texttt{op}_6$ *is the parent operator of* $\texttt{op}_5$*. The update sequence* δ*' produced by* $\texttt{op}_5$ *(Equation 15.13) is now applied to operator* $\texttt{op}_6$*, one update at a time. We first apply the update* $\texttt{u}_1$*'* $\in$ δ*'. Here* $\texttt{u}_1$*' = deleteNode (*`D'`, `E'`*). This update does not effect the operator* $\texttt{op}_6$ *(Table 15.4, Rule 1 - the condition is false). Next we apply the update* $\texttt{u}_2$*' = deleteEdge (*`C'`, `l`*) where* `l` *is the label for the edge* `e'`*:<*`C'`,`E'`*>. As* `e'` *is one of the inputs of the operator* $\texttt{op}_6$ *and the conditions in Table 15.4, Rule 2 are true, we state that the operator* $\texttt{op}_6$ *is affected by the update* $\texttt{u}_2$*'. It produces an update* $\texttt{u}_3$*' = deleteEdge (*`A'`,`l'`*) where* `l'` *is the label of the edge* `e''`*:<*`A'`,`E'`*> (Table 15.4, Rule 2). Based on the* Gen_Propagation *algorithm, as the operator* $\texttt{op}_6$ *derives from the operator* $\texttt{op}_5$*, the update sequence* δ*' produced by the operator* $\texttt{op}_5$ *is discarded. Hence the update sequence produced as a result of applying the update sequence* δ*' (Equation 15.13) on the operator* $\texttt{op}_6$ *is*

[4]We use the subscript `op3` to denote that this is the update sequence produced by op3

$$\delta_{op6} =< \mathtt{u_3}' > . \tag{15.15}$$

The final update sequence δ" produced by $\mathtt{op_6}$ *is the combination of* δ_{op3} *(Equation 15.14) and* δ_{op6}*. Thus, the final update sequence δ" is:*

$$\delta'' =< \mathtt{u_1}', \mathtt{u_3}' > . \tag{15.16}$$

Moreover, as one input of $\mathtt{op_6}$, `e':<C', E'>` *is deleted, the operator* $\mathtt{op_6}$ *is marked for deletion. As* $\mathtt{op_6}$ *is the root of the context tree* `CT3`*, the propagation terminates at* $\mathtt{op_6}$*. The update sequence δ" is the final update sequence that must be applied to the output Sangam graph* `G'`*.*

In pass `2`*, we start at the root* $\mathtt{op_6}$ *of the CAT and do an in-order traversal of the tree. As operator* $\mathtt{op_6}$ *is marked for deletion, this mark for deletion propagates to all children operators from which* $\mathtt{op_6}$ *derives (Refer to Figure 15.5). By the algorithm (Figure 15.5), the operators* $\mathtt{op_4}$, $\mathtt{op_5}$ *and* $\mathtt{op_2}$ *are marked for deletion. The operators* $\mathtt{op_1}$ *and* $\mathtt{op_3}$ *are not marked for deletion as they are shared operators (refer to Figure 15.5). At the end of the mark for deletion propagation, all operators marked for deletion are removed from the CAG.*

The updated output Sangam graph `G''` *consists of two nodes* `A'` *(output of operator* $\mathtt{op_1}$*) and* `E'` *(output of operator* $\mathtt{op_3}$*) that are disconnected. As the Sangam graph* `G''` *is not a well-formed Sangam graph (Definition 21), the update process is aborted and the Sangam graph* `G''` *rolled back to the Sangam graph* `G'`*.*

15.2.2 Propagation Rules for SAG-DU Operations

In Figures 15.6-15.9 we give the default propagation tables of the Sangam graph data update (SAG-DU) operations for the four cross algebra operators. Recall from Chapter 12 that the notation $\otimes_{o'}$(o) implies that the object o′ is produced by a cross mapping of the object o. Each update case indicates the behavior for the algebra operator when the conditions are true. If the conditions are not true there is no effect on the algebra operator, i.e., no updated sequence is generated by the operator.

SAG-DU Operation	Condition	Update Produced
addObject (v, m)	**if** $\otimes_{m'}$(m)	addObject (v, m′)
deleteObject (o, m)	**if** $\otimes_{m'}$(m) && $\otimes_{o'}$(o)	deleteObject (o′,m′)
	else	no effect
addEdgeObject (o_1,o_2, e)	no effect	no effect
deleteEdgeObject (o_e:<o_1, o_2>, e)	no effect	no effect

Table 15.6: Default SAG-DU Propagation Rules for $\otimes_{m'}$(m).

Propagation By Example: A Sangam Graph Data Update Propagation Example

Consider the CAT given in Figure 15.14. Now let the extent of the input Sangam graph G of the CAT be modified by the Sangam graph primitive: **addObject** (v, A). This operation creates a new object o with value v, and then inserts o into the extent R (A). Let u_1 = addObject (v, A) denote this update to node A. To propagate this update u_1 to the output Sangam graph G′, we begin with the first pass of the *Gen_Propagation* algorithm given in

SAG-DU Operation	Condition	Update Produced
addObject (v, m)		no effect
deleteObject (o, m)	**if** $\otimes_{m'}$(m) && $\otimes_{o'}$(o) && $\ominus_{e'}$(e) && **if** $o_{e'}$:<o_1, o_2> such that o' = o_1 or o' = o_2	deleteEdgeObject ($o_{e'}$,e'))
	else	no effect
addEdgeObject (o_1,o_2, e)	**if** $\ominus_{e'}$(e) && $\otimes_{o_1'}$ (o_1) && $\otimes_{o_2'}$ (o_2)	addEdgeObject (o_1', o_2', e')
	else	no effect
deleteEdgeObject (o_e, e)	**if** $\ominus_{e'}$(e) && $\ominus_{o_e'}$(o_e)	deleteEdgeObject (o_e', e')

Table 15.7: Default SAG-DU Propagation Rules for $\ominus_{e'}$ (e), where e':<m', n'> and e:<m, n>, and $\otimes_{m'}$(m) and $\otimes_{n'}$(n).

SAG-DU Operation	Condition	Update Produced
addObject (v, m)		no effect
deleteObject (o, m)	**if** $\otimes_{m'}$(m) && $\otimes_{o'}$(o) && $\oslash_{e'}$(e_1, e_2) && $o_{e'}$:<o_1, o_2> such that o' = o_1 or o' = o_2	deleteEdgeObject ($o_{e'}$,e')
	else	no effect
addEdgeObject (o_1,o_2, e)		no effect
deleteEdgeObject (o_e, e)	**if** o_e:<o_1,o_2 && $\oslash_{e'}$(e_1,e_2) && $\oslash_{o_e'}$(o_{e1}',o_{e2}') && e = e_1 or e = e_2	deleteEdgeObject (o_e', e')

Table 15.8: Default SAG-DU Propagation Rules for $\oslash_{e'}$ (e_1, e_2).

Figure 15.1. The update u_1 is applied to each leaf operator op_1, op_2 and op_3 of the CAT shown in Figure 15.14. By the conditions given in Table 15.6 Rule 1, only operator op_1 is affected by the update u_1 producing an update

SAG-DU Operation	Condition	Update Produced
addObject (v, m)		no effect
deleteObject (o, m)	**if** $\oslash_{e1',e2',n'}$(e), where e:<m,p>, e1':<m', n'>, and e2':<n', p'>, and $\otimes_{m'}$(m) and $\otimes_{p'}$(p) && $\oslash_{o_{e1'},o_{e2'},o_{n'}}$($o_e$)	deleteEdgeObject (o_{e1}, e1), deleteEdgeObject (o_{e2}, e2), deleteObject ($o_{n'}$, n')
addEdgeObject (o_1,o_2, e)	**if** $\oslash_{e1',e2',n'}$(e)	addObject (sys, n'), addEdgeObject (o_{e1}', e1'), addEdgeObject (o_{e2}', e2'), where sys is a system generated value
deleteEdgeObject (o_e, e)	**if** $\oslash_{e1',e2',n'}$(e), where e:<m,p>, e1':<m', n'>, and e2':<n', p'>, and $\otimes_{m'}$(m) and $\otimes_{p'}$(p) && $\oslash_{o_{e1'},o_{e2'},o_{n'}}$($o_e$)	deleteObject ($o_{n'}$, n'), deleteEdgeObject (o_{e1}', e1'), deleteEdgeObject (o_{e2}', e2')

Table 15.9: Default SAG-DU Propagation Rules for $\oslash_{e1',e2',n'}$(e).

u_1' = addObject (v, A'), where A' is the output of algebra operator op_1. Thus, the update sequence δ' produced by operator op_1 is given as:

$$\delta' =< u_1' > \tag{15.17}$$

By the *Gen_Propagation* algorithm (Figure 15.1), the update sequence δ' (Equation 15.17) produced by the operator op_1 must be propagated to its parents op_4 and op_6.

Consider first the parent operator $\mathtt{op}_6$[5]. The operator $\mathtt{op}_6$ is unaffected by both the original update $\mathtt{u}_1$ (this must be applied as the two operators are related by context dependency - refer Figure 15.1) and the update $\mathtt{u}_1{}'$ in the update sequence δ' produced by operator $\mathtt{op}_1$. As the two operators are associated by context dependency, the update sequence generated by $\mathtt{op}_6$ is concatenated with the update sequence of $\mathtt{op}_1$. Thus, the update sequence δ" produced by operator $\mathtt{op}_6$ is given as:

$$\delta'' =< \mathtt{u}_1{}' > \tag{15.18}$$

Next consider the operator $\mathtt{op}_4$. By the conditions given in Table 15.8 Rule 1, the operator $\mathtt{op}_4$ is not affected by the original update $\mathtt{u}_1$ and the update $\mathtt{u}_1{}'$ and hence produces an empty update sequence. However, as the operator $\mathtt{op}_4$ is in context relationship with the operator $\mathtt{op}_1$, the update sequence δ' produced by $\mathtt{op}_1$ must be concatenated with the final sequence of the parent operator $\mathtt{op}_4$. Hence the update sequence δ"' produced by $\mathtt{op}_4$ is:

$$\delta''' =< \mathtt{u}_1{}' > \tag{15.19}$$

Propagating this update sequence to the parent operator $\mathtt{op}_6$ of $\mathtt{op}_4$, we find that the operator $\mathtt{op}_6$ is unaffected by the update $\mathtt{u}_1{}' \in \delta$"' (by the

[5] The actual order of propagation is irrelevant.

conditions of Table 15.8, Rule 1). Moreover, as operator $\mathtt{op}_6$ derives from operator $\mathtt{op}_4$, the update sequence δ"" produced by $\mathtt{op}_6$ is empty. Hence,

$$\delta'''' =<> \tag{15.20}$$

The update propagation terminates with the operator $\mathtt{op}_6$ as $\mathtt{op}_6$ is the root of the CAT CT3. The final update sequence produced by the operator $\mathtt{op}_6$ is the concatenation of the two update sequences produced by the propagation of the update sequences of $\mathtt{op}_1$ (Equation 15.18) and operator $\mathtt{op}_4$ (Equations 15.20). Hence, the final update sequence δ_{final} is given as:

$$\delta_{final} =< \mathtt{u_1}' > \tag{15.21}$$

As no operators are marked for deletion, the second pass (Figure 15.5) will result only in the application of δ_{final} to the extent of the output Sangam graph G', i.e., in the addition of a new object o' with value v to the extent I (A') of node A'.

15.3 Properties of Propagation Through CAGs

15.3.1 Correctness of Propagation

An important property that must be satisfied by the propagation algorithms, *Gen_Propagation* and *Insert_Propagation* , is the correctness of the re-

sultant output Sangam graph . That is, the algorithms must ensure that the modified output Sangam graph $\texttt{G}_u$' produced by applying the update sequence is indeed a valid Sangam graph . For this we state the following two results.

Lemma 11 (Propagation Correctness) *The propagation of an update* u *through a cross algebra graph* CAG *produces a well-formed cross algebra graph* CAG′ *.*

Let CAG represent the original CAG that maps the input Sangam graph G to the output Sangam graph G′. Now let the input Sangam graph G be modified by the application of an update operation u.

Case 1 (u = **insertNode):** The insertNode (insertNodeAt) operation is propagated using the *Insert_Propagation* algorithm. By the *Insert_Propagation* algorithm, the propagation of the insertNode operation produces a new context dependency tree CT such that the modified CAG CAG′ is given as:

$$\texttt{CAG}' = (\texttt{CAG}) \circ (\texttt{CT}) \tag{15.22}$$

This is a well-formed CAG by definition (Definition 19).

Case 2 (u = **insertEdge):** The same as above (for insertNode) holds for the propagation of the insertEdge (and insertEdgeAt) operation using the *Insert_Propagation* algorithm.

Case 3 (u = **deleteNode):** The deleteNode operation is propagated using the *Gen_Propagation* algorithm. For this operation, we consider the following scenarios:

1. $\texttt{CAG} = \texttt{op}_i$ - Let $\texttt{CAG}$ be a single operator $\texttt{op}_i$. If the operator $\texttt{op}_i$ is affected by the update deleteNode , it would imply that the input for the operator $\texttt{op}_i$ has been deleted (Tables 15.2- 15.5). In this case the operator $\texttt{op}_i$ is deleted. Thus, the updated cross algebra graph $\texttt{CAG}$' $= \emptyset$ which is a valid cross algebra graph.

   If the operator $\texttt{op}_i$ is not affected by the update deleteNode , then the updated cross algebra graph $\texttt{CAG'} = \texttt{CAG}$, and is hence well-formed (Definition 19).

2. $\texttt{CAG} = \texttt{op}_i,\texttt{op}_j$ - Let the CAG consist of two nodes that are connected via a context dependency edge (represented by ',' in the expression). If the operators $\texttt{op}_i$ or $\texttt{op}_j$ are affected by the update deleteNode , then they are deleted, resulting in one of the following scenarios: $\texttt{CAG'} = \texttt{op}_i$, or $\texttt{CAG'} = \texttt{op}_j$, or $\texttt{CAG'} = \emptyset$.

   If the two operators are not affected by the update deleteNode , then $\texttt{CAG'} = \texttt{op}_i,\texttt{op}_j$. In all of these cases, the cross algebra graph $\texttt{CAG'}$ is a well-formed CAG (Definition 19).

3. $\texttt{CAG} = \texttt{op}_i(\texttt{op}_j)$ - Let the cross algebra graph $\texttt{CAG}$ consist of two operators $\texttt{op}_i$ and $\texttt{op}_j$ connected by a derivation edge. If the operator $\texttt{op}_j$ is affected by the update deleteNode , then this will result in the deletion of the operator $\texttt{op}_j$ and consequently the deletion of the operator $\texttt{op}_i$. In this case, $\texttt{CAG'} = \emptyset$.

   If the operator $\texttt{op}_j$ is not affected by the update deleteNode then the update is not propagated to operator $\texttt{op}_i$, i.e., $\texttt{op}_i$ is also not affected

by the update. Hence, $\texttt{CAG}' = \texttt{op}_i(\texttt{op}_j)$. In both of the cases, the final CAG is a well-formed CAG (Definition 19).

4. $\texttt{CAG} = \texttt{op}_i \circ \texttt{op}_j$ - This case can be treated in the same manner as (1).

Each of the above scenarios and the subsequent results can be extended without loss of generality to larger CAGs.

Other Cases: The cases for deleteEdge , and rename also produce well-formed CAGs. These can be argued similar to the discussions above.

The updates $\texttt{u} = \{$addObject , deleteObject , addEdgeObject , deleteEdgeObject $\}$ do not affect the structure of the CAG. Hence, if the cross algebra graph CAG is well-formed, then $\texttt{CAG}' = \texttt{CAG}$ is also well-formed after the application of an update u.

Lemma 12 (Valid Propagation) *The propagation of an update* u *through a cross algebra graph* CAG *produces a valid modified Sangam graph* $\texttt{G}_u{}'$.

This result follows directly from Lemma 11.

15.3.2 Incremental Propagation Versus Recomputation

The *Gen_Propagation* and the *Insert_Propagation* algorithms are *incremental* algorithms, i.e., an update applied on the input Sangam graph G is propagated through the CAG and applied to the output Sangam graph $\texttt{G}'$ without requiring the complete re-evaluation of the entire CAG. Both the incremental propagation and the re-evaluation of the CAG will result in the same modified output Sangam graph $\texttt{G}_u{}'$.

Theorem 11 (Incremental Propagation) *Let* `CAG`*' be a CAG produced by the application of an update* `u` *on the CAG* `CAG`*. An update sequence δ, generated by the propagation of an update* `u` *through* `CAG` *when applied to the output Sangam graph* `G'` *produces an updated Sangam graph G_u' such that G_u' $=$ G_u'', where G_u'' is the Sangam graph produced by the evaluation of* `CAG'`*.*

Let `CAG` represent a CAG, and `u` an update operation that is propagated through `CAG`. Let δ the update sequence generated by the propagation of `u` through `CAG`, and `CAG'` represent the modified CAG. Further let `out` denote the output of `CAG` and `out'` the output of `CAG'`. Thus, for incremental propagation to produce the same output as the re-computation of `CAG'`, we should have:

$$\texttt{out}' = \texttt{out} \pm \delta \tag{15.23}$$

We consider this on a case by case basis.

CASE 1 - CAG $=$ $\texttt{op}_i$**:** Here first consider that `u` $=$ deleteNode (`m`, `n`). We assume that node `m` is null to represent that node `n` is a root node.

1. $\texttt{op}_i = \otimes_{n'}$(`n`)**:** The output `out` of `CAG` here is `n'`. By Table 15.2 Rule 1 the update `u` $=$ deleteNode (`m, n`) will modify the CAG such that `CAG'` $=$. Hence, the output `out'` produced by `CAG'` is: `out'` $=$. The propagation of `u` through `CAG` will also produce the update sequence

$\delta = <$ deleteNode $(\texttt{m'}, \texttt{n'}) >$. The application of this update sequence δ on the output out of CAG is given as: out $\pm \delta = \emptyset$. In this case,

$$\texttt{out}' = \texttt{out} \pm \delta \tag{15.24}$$

2. $\texttt{op}_i = \ominus_{e'}(\texttt{e})$: This operation is not applicable to the $\ominus$ operation when n is root node (Chapter 14).

3. $\texttt{op}_i = \circleddash e'(\texttt{e1}, \texttt{e2})$: This operation is not applicable to the $\ominus$ operation when n is root node (Chapter 14).

4. $\texttt{op}_i = \oslash_{e1', e2'}(\texttt{e})$: This operation is not applicable to the $\ominus$ operation when n is root node (Chapter 14).

A similar examination with CAG $=$ $\texttt{op}_i$ and u $\in$ {insertEdge , deleteEdge , rename , addObject , deleteObject , addEdgeObject , deleteEdgeObject } shows that out' $=$ out $\pm \delta$, where out', $\hat{o}$ut, and δ are as defined above.

CASE 2: CAG = $\mathbf{op}_i \circ \mathbf{op}_j$: Here the output out of CAG $= \texttt{out}_i \bigcup \texttt{out}_j$, where $\texttt{out}_i$ and $\texttt{out}_j$ are the outputs of the operations $\texttt{op}_i$ and $\texttt{op}_j$ respectively. Let u = deleteNode (m, n) denote the update. Further let $\texttt{op}_i$ be affected by u. Let CAG' denote the CAG after the propagation of u, and out' the output of CAG'. We thus have:

$$\texttt{out}' = \texttt{out}_\texttt{i}' \bigcup \texttt{out}_\texttt{j}. \tag{15.25}$$

The operation $\mathtt{op}_j$ here is unaffected. By CASE 1, we know that if CAG $= \mathtt{op}_i$, then $\mathtt{out}' = \mathtt{out} \pm \delta$. Thus, we have $\mathtt{out}_i' = \mathtt{out}_i \pm \delta$. Replacing $\mathtt{out}_i'$ in Equation 15.25, we have:

$$\mathtt{out}' = (\mathtt{out_i} \pm \delta) \bigcup \mathtt{out_j}. \tag{15.26}$$

This result can be shown for the case when $\mathtt{op}_j$ is affected or if both $\mathtt{op}_i$ and $\mathtt{op}_j$ are affected. This result can also be extended without loss of generality to CAGs.

CASE 3: CAG = $\mathbf{op}_i$, $\mathbf{op}_j$: Here the output out of CAG is given as $\mathtt{out} = \mathtt{out}_i \bigcup \mathtt{out}_j$, where $\mathtt{out}_i$ and $\mathtt{out}_j$ are the outputs of the operations $\mathtt{op}_i$ and $\mathtt{op}_j$ respectively. Recall that the symbol "," here reflects a context dependency relationship. Let u denote the update applied to CAG. Further let $\mathtt{op}_i$ be affected by u. Let CAG' denote the modified CAG with output out'. Thus, $\mathtt{out}' = \mathtt{out}_i' \bigcup \mathtt{out}_j$. Based on CASE 1 and similar to CASE 2, we can replace $\mathtt{out}_i'$ with $\mathtt{out}_i \pm \delta$, where $\delta = < \mathtt{u}' >$ and u' is the update produced by $\mathtt{op}_i$. Thus incremental propagation achieves the same resultant output as recomputation. A similar argument can be made if only operator $\mathtt{op}_j$ is affected by the update u.

Now let us consider that both operators $\mathtt{op}_i$ and $\mathtt{op}_j$ are affected by the update u such that $\delta = <\mathtt{u1}', \mathtt{u2}'>$, where $\mathtt{u}_i'$ and $\mathtt{u}_j'$ are the updates generated by $\mathtt{op}_i$ and $\mathtt{op}_j$ respectively. Breaking down δ into individual updates, and based on CASE 1 we have:

$$\mathtt{out_i}' = \mathtt{out_i} \pm \mathtt{u_i}' \tag{15.27}$$

$$\mathtt{out_j}' = \mathtt{out_j} \pm \mathtt{u_j}' \tag{15.28}$$

We know by definition (Definition 19), that `out'` of the modified CAG `CAG`' when both $\mathtt{op}_i$ and $\mathtt{op}_j$ are affected is given as:

$$\mathtt{out}' = \mathtt{out_i}' \bigcup \mathtt{out_j}' \tag{15.29}$$

Replacing $\mathtt{out}_i$' and $\mathtt{out}_j$' in Equation 15.29 using Equation 15.27 and Equation 15.28, we have:

$$\mathtt{out}' = (\mathtt{out_i} \pm \mathtt{u_i}') \bigcup (\mathtt{out_j} \pm \mathtt{u_j}') \tag{15.30}$$

$$\mathtt{out}' = \mathtt{out_i} \bigcup \mathtt{out_j} \pm \mathtt{u_i}' \pm \mathtt{u_j}'$$

$$\mathtt{out}' = \mathtt{out_i} \bigcup \mathtt{out_j} \pm \delta$$

This result can be generalized to CAGs of arbitrary size.

CASE 4: CAG $=$ op$_i$(op$_j$): Here the output `out` of `CAG` is given as `out` $=$ $\mathtt{out}_i$, where $\mathtt{out}_i$ is the output of the outer most operation $\mathtt{op}_i$. Recall that the symbol "( )" here reflects a derivation relationship. Also recall that

an outer operation is affected by an update only if its leaves are affected by the update. Thus, let us assume that operation $\texttt{op}_j$ is affected by the update $\texttt{u}$. Let $\delta = <\texttt{u'}>$ be the updated generated by $\texttt{op}_j$. We know by the ***Gen_Propagation*** propagation algorithm that only updates generated by the children operators are applied to the parent operator. Hence, we apply the update $\texttt{u'}$ on the operator $\texttt{op}_i$. The original update $\texttt{u}$ is not applied to $\texttt{op}_i$. Let δ' $= <\texttt{u''}$ denote the update generated by $\texttt{op}_i$. By CASE 1, we have:

$$\texttt{out}_\texttt{i}{}' = \texttt{out}_\texttt{i} \pm \texttt{u}'' \tag{15.31}$$

This result can also be generalized to CAGs of arbitrary size.

The Insert Operations. We next consider the insertNode and insertEdge operations. Let $\texttt{CAG} = \texttt{CAG}_1 \circ \texttt{CAG}_2 \ldots \circ \texttt{CAG}_i$ be a CAG. Let the input SAG $\texttt{G}$ be modified by the update $\texttt{u} \in \{$ insertNode , insertNodeAt , insertEdge , insertEdgeAt $\}$. We know by definition (Definition 19), the output of $\texttt{CAG}$ is given as: $\texttt{out} = \texttt{out}_1 \bigcup \texttt{out}_2 \bigcup \ldots \texttt{out}_\texttt{i}$. By the ***Insert_Propagation*** algorithm, the propagation of these operations always produces a new context dependency tree $\texttt{CT}_{j(out'_j)}$. The modified CAG is given as:

$$\texttt{CAG}' = \texttt{CAG}_1 \circ \texttt{CAG}_2 \ldots \circ \texttt{CAG}_\texttt{i} \circ \texttt{CT}_\texttt{j} \tag{15.32}$$

The output of CAG' is given as:

$$\mathtt{out}' = \mathtt{out}_1 \bigcup \mathtt{out}_2 \ldots \bigcup \mathtt{out}_\mathtt{i} \bigcup \mathtt{out}'_\mathtt{j} \tag{15.33}$$

This can be re-written as:

$$\mathtt{out}' = \mathtt{out} \bigcup \mathtt{out}'_\mathtt{j} \tag{15.34}$$

where out is the output of CAG. Thus incremental propagation produces the same result as re-computation. □

15.4 Summary

In this chapter, we have presented an incremental propagation algorithm that can propagate the updates as presented in Chapter 14 through a cross algebra expression, to produce a set of updates that can be applied on the output Sangam graph. We have presented default propagation strategies for each algebra operator and update; and have shown that incremental propagation is equivalent to full recomputation. However, we should point out that if the original cross algebra expression is built based on rules such as the basic or shared inlining rules, then the default propagation rules for the *insert* evolution operations cannot guarantee that the rules used for the

construction of the cross algebra expression will be preserved after default propagation.

Chapter 16

Performance Comparison: Incremental vs Re-computation

One of the key motivations for incremental propagation is the benefit to performance, measured in terms of the total time taken to produce the updated output, over re-computation. In this chapter, we present experimental results to: (1) show that incremental propagation has lower execution times than re-computation; and (2) analyze the performance of the incremental propagation for different cross algebra operators and different types of cross algebra graphs.

All experiments used `personal.dtd`, represented in Figure 16.1 as the input. The *xmlPGen* tool was used to generate the corresponding XML documents with a uniform distribution of the number of objects that are instances of the nodes and edges of the Sangam graph. The cross algebra graphs (CAGs), *ident* and *inline*, and the different cross algebra operators

were constructed as described in Section 13.2. All data update operations involved the `person`, `name` and `given` elements. All data update operations were generated using a simple program that we wrote.

```
<?xml encoding="UTF-8"?>
<!ELEMENT personnel (person)+>
<!ELEMENT person (name,email*,url*,link?)>
<!ATTLIST person id ID #REQUIRED>
<!ATTLIST person note CDATA #IMPLIED>
<!ATTLIST person salary CDATA #IMPLIED>

<!ELEMENT name (family,given)>

<!ELEMENT family (#PCDATA)>

<!ELEMENT given (#PCDATA)>

<!ELEMENT email (#PCDATA)>

<!ELEMENT url EMPTY>
<!ATTLIST url href CDATA 'http://'>

<!ELEMENT link EMPTY>
<!ATTLIST link manager IDREF #IMPLIED>
<!ATTLIST link subordinates IDREFS #IMPLIED>

<!NOTATION gif PUBLIC '-//APP/Photoshop/4.0'
 'photoshop.exe'>
```

Figure 16.1: The `personal.dtd`, used as the input DTD.

The re-computation time corresponds to the time taken to evaluate the given cross algebra graph after a change to the input graph and does not include the time required to build the cross algebra graph.

In all of the experiments below, we report the propagation times for various data update operations, i.e., `insertObject`, `deleteObject`, `in-`

`sertEdgeObject` and `deleteEdgeObject`. We do not consider any schema change operations. Typically, a schema change operation has (1) some processing cost (`pc`), i.e., the cpu time taken to actually perform the change once all data is in memory; and (2) some I/O cost (`rw`), i.e., the time required to read and write the data on which the change is applied to and from the disk. The cost of the schema change operations is largely dominated by this I/O cost. This I/O cost is dependent on the particular DBMS system that was used and not on the middle-layer, i.e., the Sangam layer. The processing cost, `pc`, is the only cost affected by the propagation algorithm, the focus of this research. For schema changes this processing cost is comparable to those reported for the data update operations but is scaled according to the number of objects associated with the input node to which the change is applied.

All experiments were conducted on a Pentium IV, 933MHz, 256MB RAM system running Debian Linux, kernel version 2.2.19, using the 𝔖𝔞𝔫𝔤𝔞𝔪 prototype system described in Section 7.1. The 𝔖𝔞𝔫𝔤𝔞𝔪 system itself was built using the Java JDK version 1.3.1.

Pilot tests on our system showed that the time taken to incrementally propagate the data change operations through a CAG were from `5` to `10` milliseconds. As typical operating systems clocks have a granularity of about 10 milliseconds, we record the time taken to propagate `1000` change operations. Average per operation measurements can be easily obtained by dividing the reported results by `1000`. In the graphs presented here we report the time for `1000` change operations for both re-computation and incremental propagations.

16.1 Incremental vs. Re-Computation

Figures 16.2, 16.3 and 16.4 compare the time taken to incrementally propagate different data update operations through the `cross`, `connect` and `smooth` operators, with the time taken to re-compute the output Sangam graph for the same change. For a `cross` operator the CAG consisted of only one operator. For the `connect` and `smooth` operators the CAG included the `connect` and `smooth` operators and the requisite `cross` operators. The change `c` however was applied to only one operator (`cross`, `connect` and `smooth` respectively for the three cases). The time taken $\mathtt{T}_i$ to incrementally propagate a change `c` through a CAG `G` can be divided into two main components: (1) the time taken to do the actual propagation `pt`; and (2) the time taken to apply the set of updates produced by the propagation to the output Sangam graph `at`. Thus the total time for incremental propagation is $\mathtt{T}_i = \mathtt{pt} + \mathtt{at}$. For re-computation, the total time $\mathtt{T}_r$ is the time taken to evaluate the CAG after each change has been applied.

In Figures 16.2- 16.4, we kept the DTD (`personal.dtd`) and the operator `op` ($\mathtt{op} \in \{\mathtt{cross}, \mathtt{connect}, \mathtt{smooth}\}$) constant. We increased the size of the XML documents, i.e., the extent of the Sangam graph. To better compare all the performance of the various operations with re-computation, we show all the graphs using a logarithmic scale. Each data point here represents the average of three runs. As can be seen from the figures, the cost of re-computation is clearly higher than the cost of incrementally propagating data update operations in all cases.

We also performed a similar set of experiments for larger cross algebra

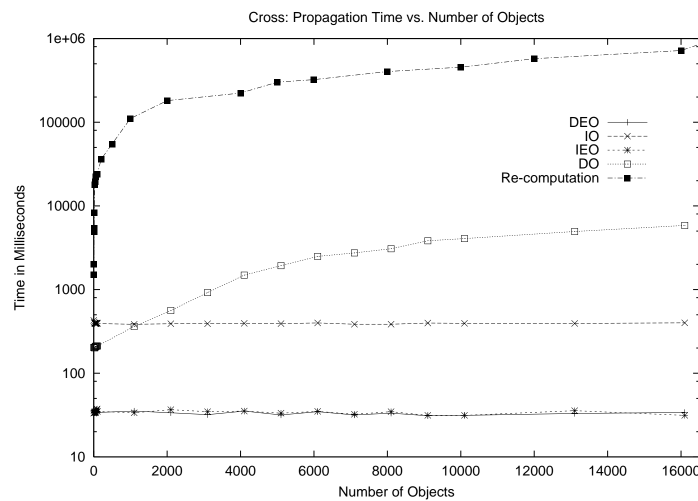


Figure 16.2: Time Taken to Incrementally Propagate the `insertObject`(IO), `deleteObject`(DO), `insertEdgeObject`(IEO) and `deleteEdgeObject`(DEO) Operations Through One Cross Algebra Operator Compared to the Cost of Re-computation.

graphs. Here, we considered the `ident` and the `inline` CAGs generated for the `personal.dtd`. Once again, we kept the DTD and structure of the CAG constant, and we increased the XML size, i.e., the extent size of the input Sangam graph. Similar to the graphs shown in Figure 16.2– 16.4, the cost for re-computation was clearly more expensive than the cost of incremental propagation in both cases as shown in Figures 16.5 and 16.6.

16.2 Analyzing the Performance of Incremental Propagation

We next analyze the cost of propagating a data update operation through the `cross`, `connect` and `smooth` algebra operators, as well as the `ident`

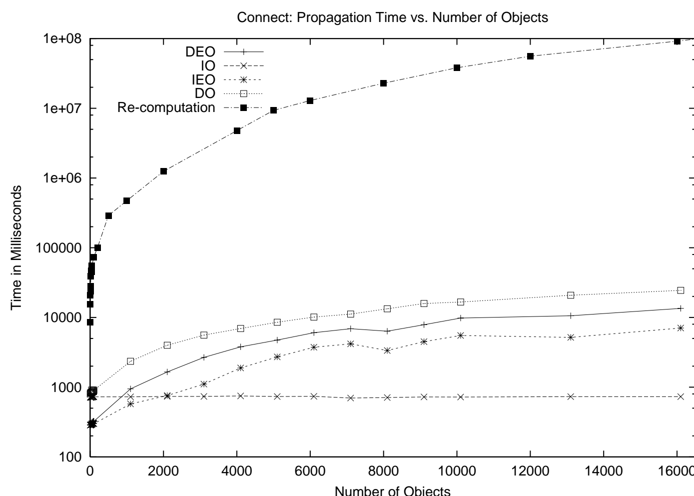


Figure 16.3: Time Taken to Incrementally Propagate the `insertObject` (IO), `deleteObject`(DO), `insertEdgeObject`(IEO) and `deleteEdgeObject`(DEO) Operations Through One Connect Algebra Operator Compared to the Cost of Re-computation.

and the `inline` CAGs.

16.2.1 The Algebra Operators

Cross Algebra Operator

Figure 16.7 depicts the time taken to propagate the different data update operations, `insertObject`, `deleteObject`, `insertEdgeObject` and `deleteEdgeObject`, through a cross algebra operator. Here we kept the input DTD and the cross algebra operator constant, and increased the number of input objects. Each data point represents the average of three runs.

The `insertObject` inserts an object into the Sangam graph. The cost to perform this operation is constant and does not vary with the input ex-

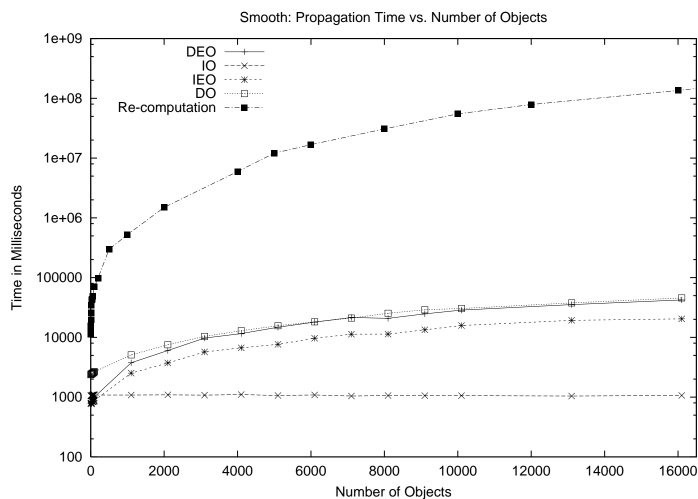


Figure 16.4: Time Taken to Incrementally Propagate the `insertObject`(IO), `deleteObject`(DO), `insertEdgeObject`(IEO) and `deleteEdgeObject`(DEO) operations Through One Smooth Algebra Operator Compared to the Cost of Re-computation.

tent size. Hence, the application time `at` is constant. Moreover, as the `cross` algebra operator is kept constant, the propagation time `pt` is also constant. This is depicted by the horizontal line depicting the propagation of the `insertObject` in Figure 16.7.

The `deleteObject` operation deletes the object with the specified value from the extent of the specified node. To apply this operation, we must first find the correct object in the extent prior to constructing the update to be applied on the output. The cost of searching for the correct object, and hence the propagation time, `pt`, increases linearly with a linear increase in the extent size. The application cost `at` is constant. As depicted in Figure 16.7, the total time for propagating the `deleteObject` operation

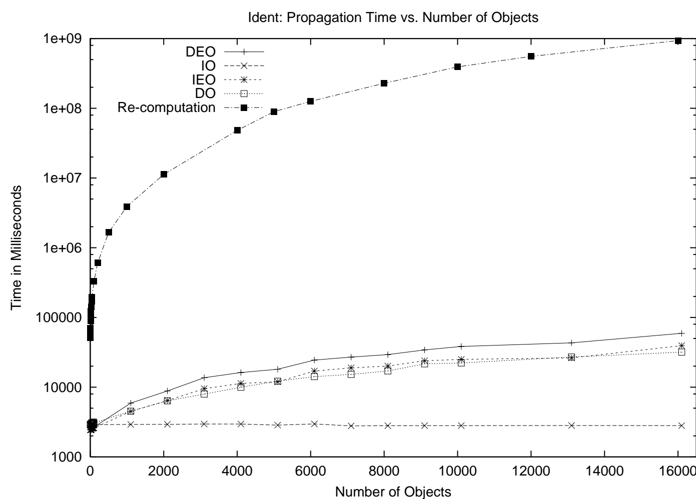


Figure 16.5: Time Taken to Incrementally Propagate the `insertObject`(IO), `deleteObject`(DO), `insertEdgeObject`(IEO) and `deleteEdgeObject`(DEO) Operations Through an `Ident` CAG Compared to the Cost of Re-computation.

through a cross algebra operator thus increases linearly with the number of objects in the extent.

As can be seen in Figure 16.7, the `insertEdgeObject` and `delete-EdgeObject` have a negligible cost since these operators are not applicable to `cross` operator (Section 15.2).

Connect Algebra Operator

Figure 16.8 depicts the time taken to propagate the different data update operations, `insertObject`, `deleteObject`, `insertEdgeObject` and `deleteEdgeObject`, through a connect algebra operator. Here, we also kept the input DTD and the connect algebra operator constant, and in-

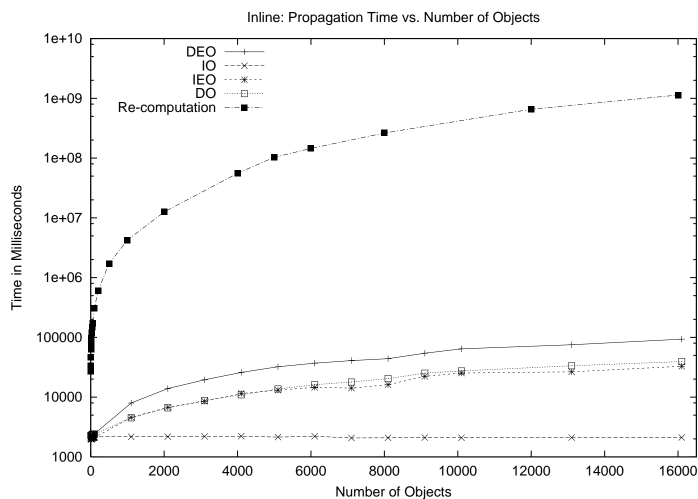


Figure 16.6: Time Taken to Incrementally Propagate the `insertObject`(IO), `deleteObject`(DO), `insertEdgeObject`(IEO) and `deleteEdgeObject`(DEO) Operations Through an `Inline` CAG Compared to the Cost of Re-computation.

creased the number of input objects. Each data point represents the average of three runs.

The `insertObject` operation is not applicable to the `connect` algebra operator (refer Table 15.7). Hence, the application time `at` = `0`. The propagation time `pt`, and hence the total time `T`, is constant as depicted in Figure 16.8.

The propagation time `pt` for the `deleteObject`, `deleteEdgeObject` and `insertEdgeObject` operations increases linearly with the increase in the extent size. Once the set of updates are produced the amount of time taken to apply one update `at` is constant. Hence, the total time `T` = `pt` + `C`, where `C` is some constant value for applying the change for all the

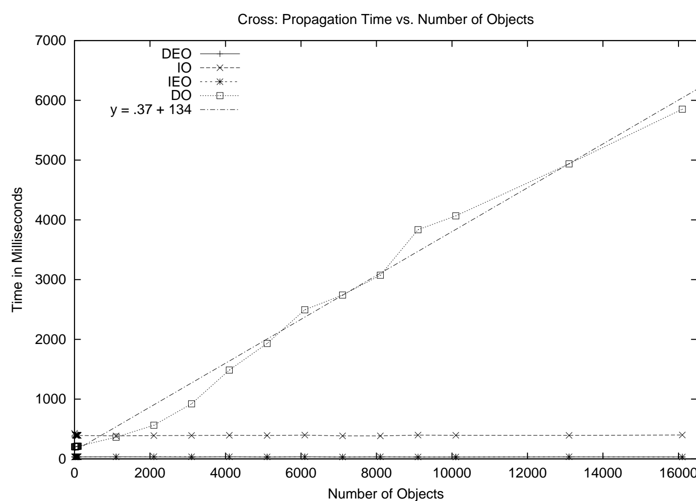


Figure 16.7: Time Taken to Incrementally Propagate the `insertObject`(IO), `deleteObject`(DO), `insertEdgeObject`(IEO) and `deleteEdgeObject`(DEO) Operations Through the `cross` Algebra Operator.

operations, increases linearly with the increase in the extent size of the input. The variation in the slope of the lines is largely due to the cost of the searches that must be performed to find the affected objects in the output Sangam graph and construct the correct update operation during the propagation. For example, for a `deleteObject` operation we must find all the edges that have the specified object `o` on either end of the edge (see Table 15.7), and then construct the correct `deleteEdgeObject` operations. In contrast, while the `insertEdgeObject` has considerable propagation cost, it will produce only one update in the case of the `connect` operator, accounting for the slower increase in the update propagation times.

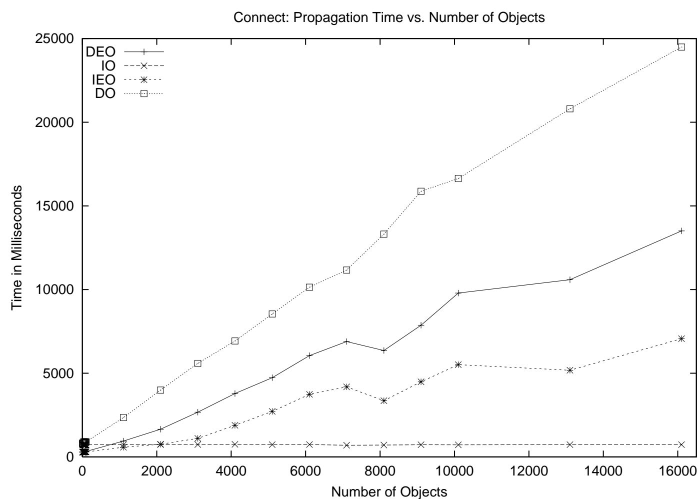


Figure 16.8: Time Taken to Incrementally Propagate the `insertObject`(IO), `deleteObject`(DO), `insertEdgeObject`(IEO) and `deleteEdgeObject`(DEO) Operations Through one `connect` Algebra Operator.

Smooth Algebra Operator

Figure 16.9 depicts the time taken to propagate the different data update operations, `insertObject`, `deleteObject`, `insertEdgeObject` and `deleteEdgeObject`, through a smooth algebra operator. Here, we once again keep the input DTD and the smooth algebra operator constant, and increase the number of input objects. Each data point represents the average of three runs.

As in the case of the `connect` operator, the `insertObject` is not applicable to the `smooth` algebra operator (see Table 15.8). Hence, the application time `at = 0`. The propagation time `pt`, and thus the total time `T`, is constant as depicted by the horizontal line in Figure 16.9.

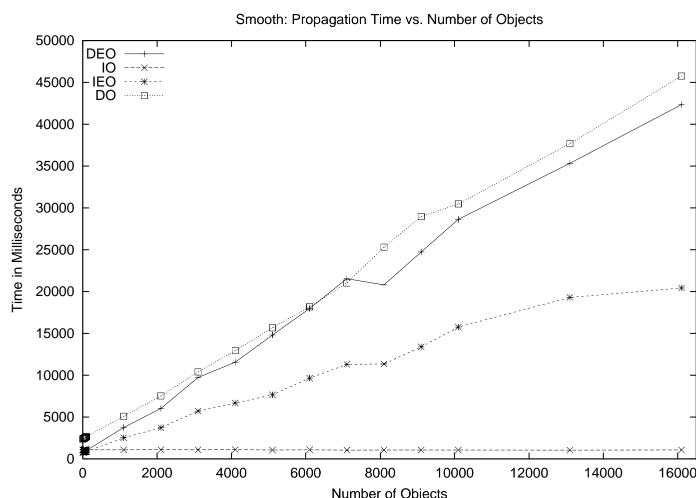


Figure 16.9: Time Taken to Incrementally Propagate the `insertObject`(IO), `deleteObject`(DO), `insertEdgeObject`(IEO) and `deleteEdgeObject`(DEO) Operations Through One Smooth Algebra Operator.

The propagation times `pt` for the `deleteObject`, `deleteEdgeObject` and `insertEdgeObject` operations increases linearly with the increase in the extent size. The variation in the slope of the total times for the `deleteObject`, `deleteEdgeObject` and the `insertEdgeObject` are as before due to the increasing search space to locate the correct objects in the output Sangam graph and construct the required updates.

Comparison of Propagation Times for Different Operators

In summary, the propagation of the data update operations through the different algebra operators depict similar trends. As expected for the `cross` operator, the cost of incrementally propagating the `insertEdgeObject`

and `deleteEdgeObject` operations is a negligible constant as these operations do not have any effect on the cross algebra operator. The `insert-Object` operation during incremental propagation is always accomplished in constant time. The propagation cost for `deleteObject` operation shows a linear increase with a linear increase in the extent size, i.e., the search space for locating the to be deleted object. Similar results hold for the other operators, `connect` and `smooth`. We found for the `connect` and `smooth` operators, that the delete operations `deleteObject` and `deleteEdge-Object` tended to be more expensive than the insertion operations `insert-EdgeObject`; and the cost of the `insertObject` operation is a negligible constant as the operation is not applicable to these operators.

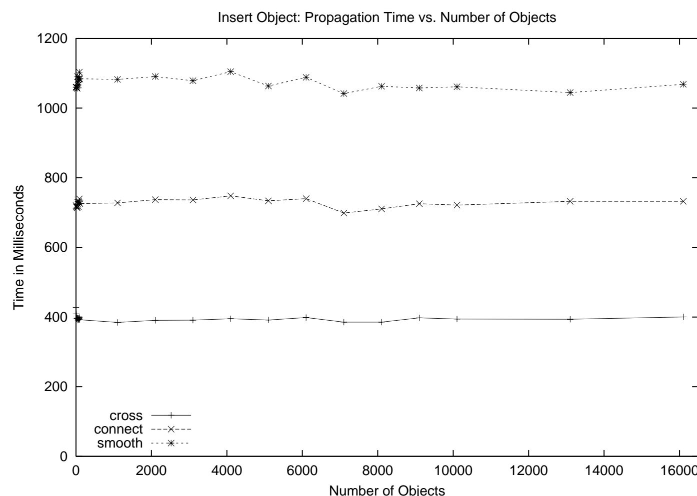


Figure 16.10: Time Taken to Incrementally Propagate the `insertObject` Operation Through one `cross`, `connect` and `smooth` Algebra Operators.

Figures 16.10- 16.13 depict the cost of each data update through the different algebra operators. In all cases, the cost of propagating any data up-

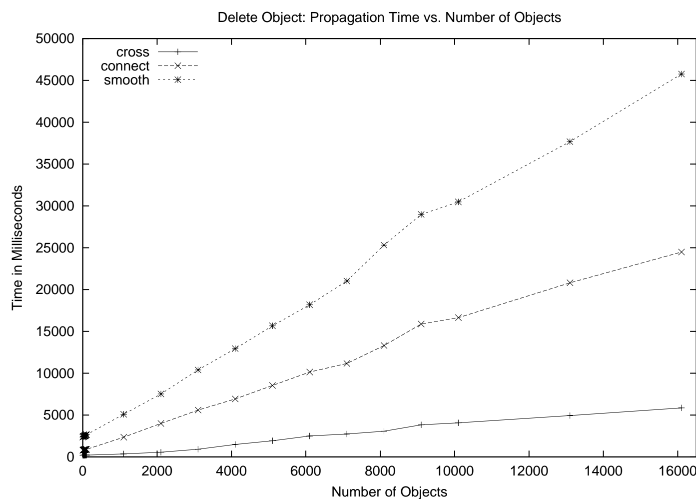


Figure 16.11: Time Taken to Incrementally Propagate the `deleteObject` Operation Through one `cross`, `connect` and `smooth` Algebra Operators.

date through the `smooth` operator is higher than the cost of propagating the same update through the `connect` and `cross` operators. This is primarily because each operation requires more number of the searches when applied to the `smooth` operator, thereby explaining the elevated costs.

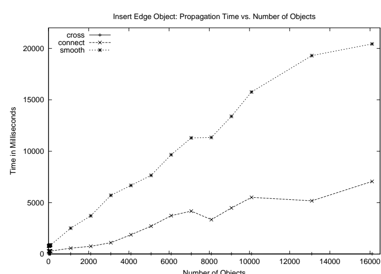


Figure 16.12: Time Taken to Incrementally Propagate the `insertEdgeObject` Operation Through one `cross`, `connect` and `smooth` Algebra Operators.

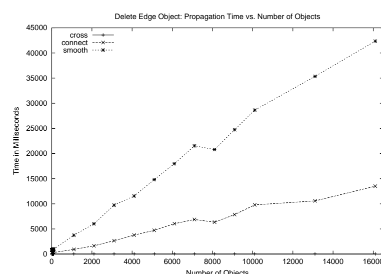


Figure 16.13: Time Taken to Incrementally Propagate the `deleteEdgeObject` Operation Through one `cross`, `connect` and `smooth` Algebra Operators.

16.2.2 Ident and Inline CAGs

Variation in Extent Size

Figures 16.14 and 16.15 depict the cost of propagating the different data updates through an `ident` and an `inline` CAG. For each experiment, we kept the input DTD and the CAG structure constant, and linearly increased the size of the input XML documents, i.e., the number of objects in the input extent. Each data point represents the average of three runs. Overall, we find that the performance of the `insertObject` operation in both cases is a constant. The propagation costs for the other three operations, `deleteObject`, `deleteEdgeObject` and `insertEdgeObject`, increases linearly with the linear increase in the extent size. The performance costs for the `deleteObject` and the `deleteEdgeObject` were once again higher than those for the `insertEdgeObject` operation due to the increased search spaces for constructing the updates.

Figures 16.16- 16.18 compare the cost of propagating the different update operations for the `ident` and `inline` CAGs. In all cases, the cost of propagating the data updates through the `inline` CAG are higher than the cost of propagating the same change through the `ident` CAG. The `ident` CAG consists of `cross` and `connect` operators, that form many separate, smaller `CATs` with a large number of shared operators. On the other hand, the `inline` CAG contains fewer number of algebra operators, but these operators are some composition of `cross`, `connect` and `smooth` operators. The number of shared operators, in general, is fewer than in the `inline` CAG. The performance of the propagation algorithm is not af-

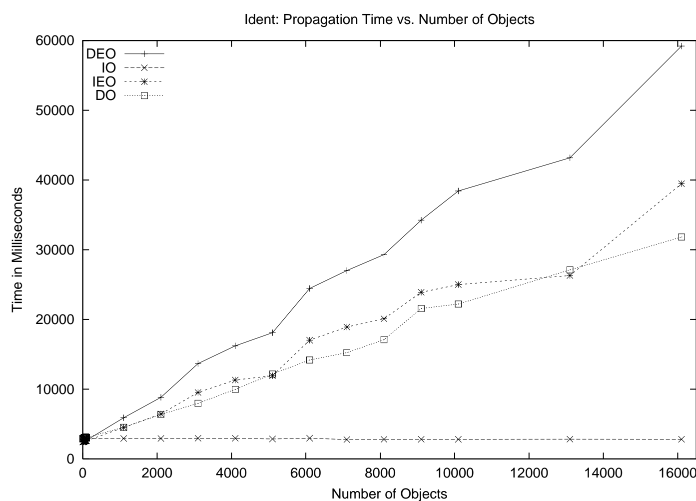


Figure 16.14: Time Taken to Incrementally Propagate the `insertObject`, `deleteObject`, `insertEdgeObject` and `deleteEdgeObject` Operations Through the `ident` CAG.

fected by the number of shared operators, as each must still be examined in the context of new parents to determine the overall effect of the update. We thus attribute the difference in the performance of the update propagation to the fact that the `inline` CAG contains `smooth` operators which are expensive for propagation.

Variation in Structure Size

To examine the effects of the CAG size on the performance of the update propagation algorithm, we also conducted some experiments that varied the size of the DTD, i.e., the number of elements in the DTD, causing a relative increase in the size of the resultant CAG. For this set of experiments, we kept the extent size of the XML documents (and hence the extent size

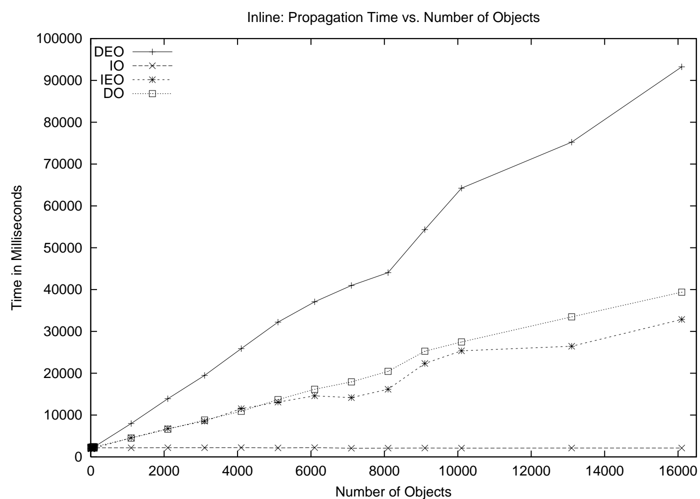


Figure 16.15: Time Taken to Incrementally Propagate the `insertObject`, `deleteObject`, `insertEdgeObject` and `deleteEdgeObject` Operations Through the `inline` CAG.

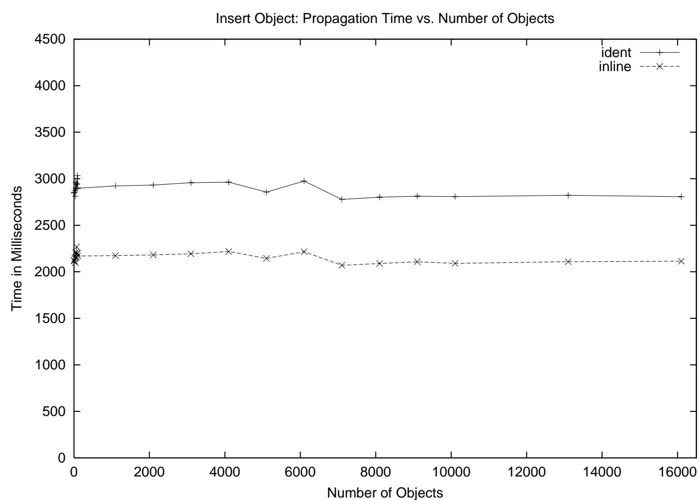


Figure 16.16: Time Taken to Incrementally Propagate the `insertObject` (IO) Operation Through the `ident` and `inline` CAG.

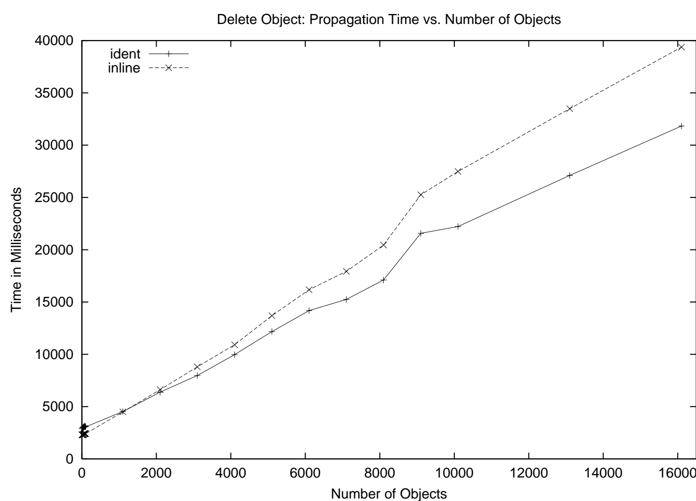


Figure 16.17: Time Taken to Incrementally Propagate the `deleteObject`(DO) Operation Through the `ident` and `inline` CAG.

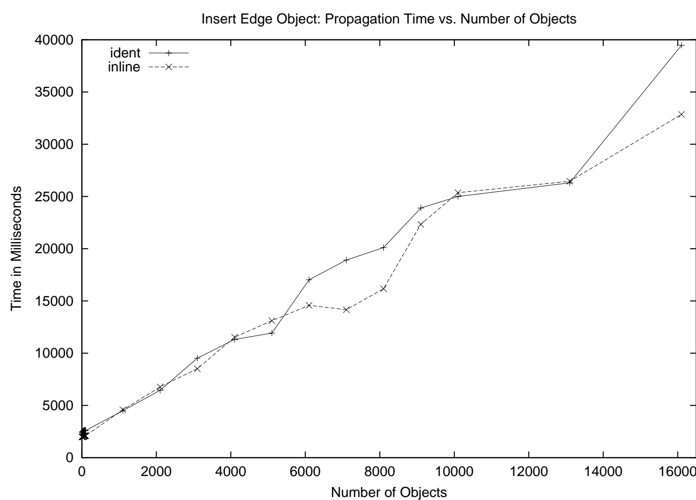


Figure 16.18: Time Taken to Incrementally Propagate the `insertEdgeObject`(IEO) Operation Through the `ident` and `inline` CAG.

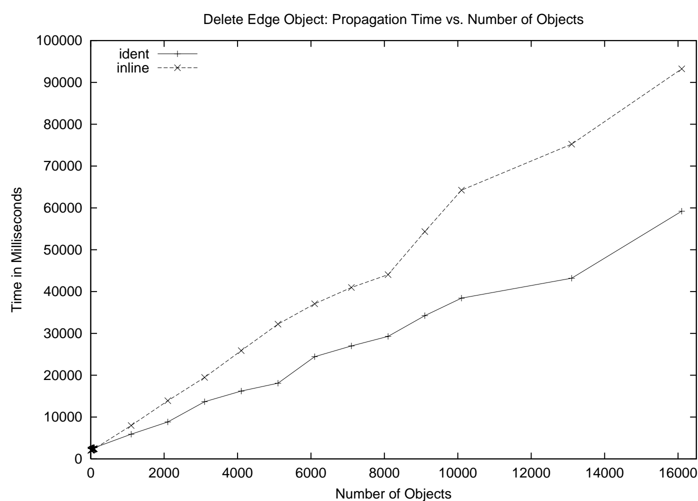


Figure 16.19: Time Taken to Incrementally Propagate the `deleteEdgeObject`(DEO) Operation Through the `ident` and `inline` CAG.

of the Sangam graph) constant at a uniform distribution of `10000` objects, and varied the DTD and consequently the structure of the CAG. Each data point represents the average of three runs.

Figures 16.20- 16.23 show the results of our experiments. In general, we found that the performance of the update propagation algorithm for each data update increased with a linear increase in the number of operators in the CAG. This result holds for both `ident` and `inline` CAGs. The increase in the propagation costs for the `insertObject` and `delete-Object` operations was very small; and the propagation costs for the `in-sertEdgeObject` and `deleteEdgeObject` operations showed a logarithmic increase with a linear increase in the CAG size. While the trends for the propagation of each operation were similar for both `ident` and `inline`

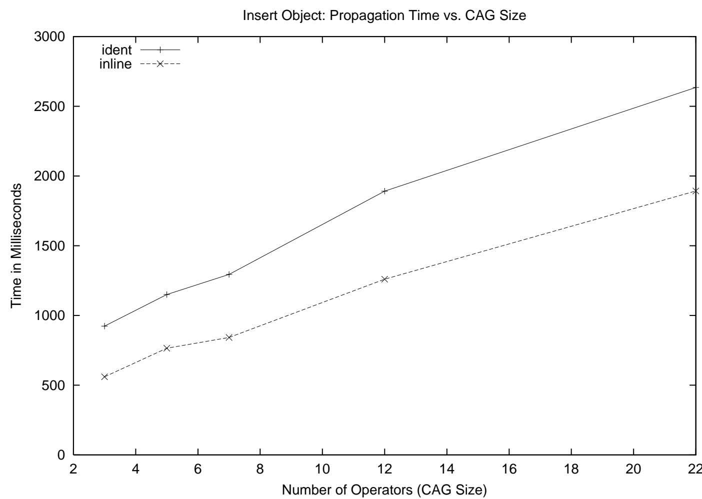


Figure 16.20: Time Taken to Incrementally Propagate the `insertObject` Operation Through the `ident` and `inline` CAG while Varying the Number of Algebra Operators in the CAG.

CAG, a surprising finding was the fact that the performance of the update propagation algorithm for an `ident` CAG degrades faster than than the performance of the `inline` CAG, leading us to believe that the change in the CAG size tends to affect the `ident` CAG more than the `inline` CAG.

16.3 Summary

To summarize, based on the experimental results we find that:

- re-computation cost is in general more expensive that the incremental propagation cost;
- the cost of re-computation increases polynomially with a linear increase in the extent size of the CAG.

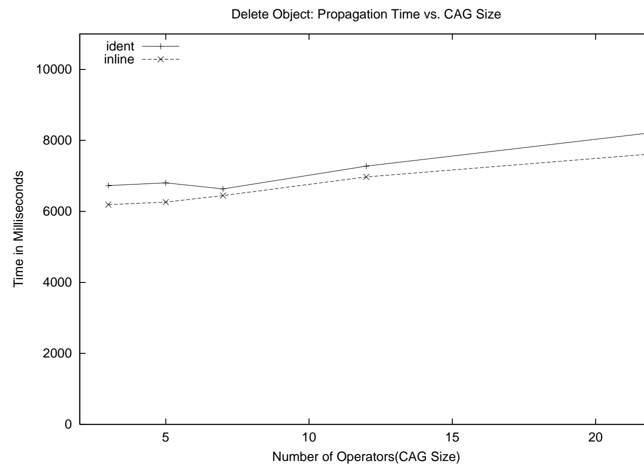


Figure 16.21: Time Taken to Incrementally Propagate the `deleteObject` Operation Through the `ident` and `inline` CAG while Varying the Number of Algebra Operators in the CAG.

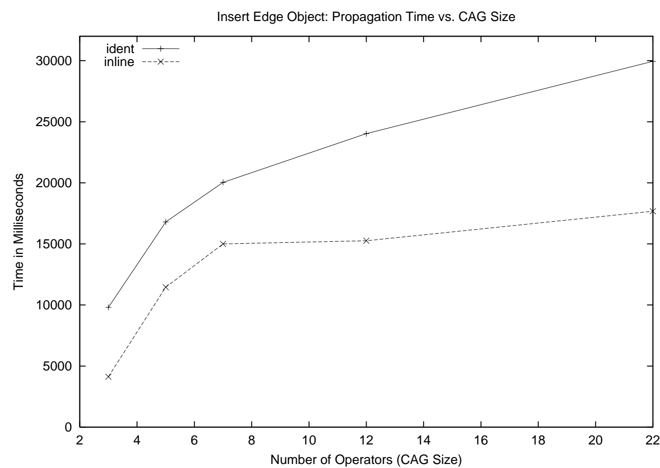


Figure 16.22: Time Taken to Incrementally Propagate the `insertEdgeObject` Operation Through the `ident` and `inline` CAG while Varying the Number of Algebra Operators in the CAG.

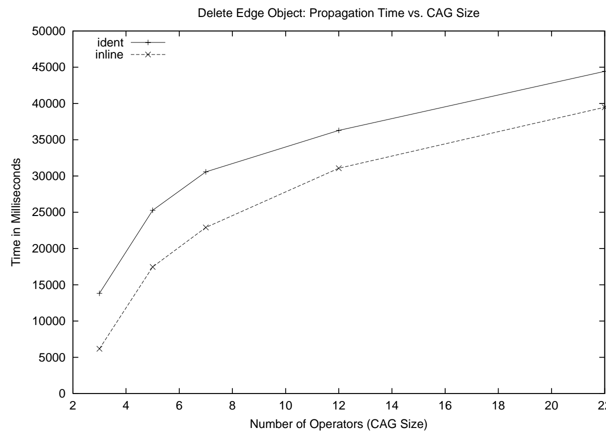


Figure 16.23: Time Taken to Incrementally Propagate the `deleteEdgeObject` Operation Through the `ident` and `inline` CAG while Varying the Number of Algebra Operators in the CAG.

- the cost of propagation for the operation `insertObject` is constant for all algebra operators;
- the cost of propagation for the all updates increases linearly with linear increase in extent size for all CAGs;
- the cost of propagation for updates `deleteEdgeObject` and `insertEdgeObject` increases logarithmically with a linear increase in CAG size; while the cost of propagation for updates `deleteObject` and `insertObject` show a negligible increase in performance cost with a linear increase in CAG size.

Chapter 17

Related Work

17.1 XML Storage

Currently there are numerous projects that deal with the persistent storage of XML documents [ZLMR01, FK99, SHT$^+$99, CFLM00]. Shanmugasundaram et al. [SHT$^+$99] presented the first storage of XML documents with DTD in a relational system and cataloged the problems and limitations of relational systems in doing so. In the same vein, Florescu et al. [FK99] in their work present eight mapping schemes for storing XML documents in relational databases with experimental data presented to help select an ideal mapping scheme. Zhang et al. [ZLMR01] present one fixed mapping for storing XML documents (with DTD) in relational systems. Catania et al. [CFLM00] present the use of object databases to efficiently store XML documents with DTDs. With the exception of the *Clock* project [ZLMR01], none of these projects look at the maintenance of the relational or the object storage when the source XML/DTD is modified. Zhang et al. [ZLMR01]

address this problem but only for their one fixed mapping. In our work, we now provide an algebra that allows us to not only map XML documents to relational, extended relational or object databases, but to also propagate any DTD change to the same independent of the mapping.

17.2 Change Management

17.2.1 Change Propagation within A Data Model

In the relational database framework, much work has been done on the update propagation from the view relations to the base relations [BS81, DB78, Kel82] and maintenance of views [ZGMHW95, GB95]. Most of these algorithms recompute the relational select-project-join views. Keller [Kel82] proposed a methodology for translating updates against a view to updates against the database. He proposed five criteria which must be satisfied by all `join` translations. His work probably comes the closest to our work in terms of translation of change from a view to the base. However, while he looked at data updates in the relational model, we focus on the translation of schema updates in the object-oriented context. In view maintenance work, Gupta et al. [GB95] proposed an algorithm for the maintenance of a view defined using a deductive query language.

Object-oriented views are largely categorized as object preserving views or object generating views. Bertino et al. [BCGMG97] have proposed a formal framework for the definition of views. However, while they briefly mention that update propagation is being investigated, this work is really not reported in the paper. From the literature survey most work on up-

date of views in the OO context has been done on object-preserving views [SLT91, San95, KR98]. In [GGMS97], Gluche et al. have proposed the incremental maintenance of object views. Their work uses algebraic properties of the materialized functions to decide if incremental update propagation is feasible. They regard update operations as monoids and any update that can be translated into a monoid is applicable for propagation. In [AYBS97], Amer-Yahia et al. present an update propagation mechanism for views defined in O_2. This mechanism extends the view definition language to allow the user to specify both the propagation as well as code that can be executed when adding, deleting or modifying instances at the base class. For example, the creation of a new object in a class `Adult`, a view defined over a general class `Person` can be represented as follows in `O2`:

```
on create in class Adult with (name:String,
                                          home:Address)
do {Persons += set(self->root-object);}
enddo;
```

Here, the class `Person` is identified as the base object to which the change is propagated. In this example, the default rules for creation at the base are utilized. However, OQL statements can be supplied as part of the `do` clause to handle specific code.

In the context of schema changes through views, [RR97] have used views as a mechanism to hide schema changes in the underlying database from the applications that are defined on it. In doing so, this mechanism allows the user to submit schema changes against a view which then need to

be propagated down to the base class as well as from the base to the view. Other than this work, Breche et al. [BFK95] have proposed using views for simulating schema changes. They look at simulating high-level primitives (complex changes) [Bré96] using an enhanced O_2 view mechanism. A user can specify all changes and these are translated into schema propagation changes when the user is ready to materialize their changes. The views used for simulation are materialized views. Thus the base data is *migrated* towards the data in the simulated view. This migration is when beyond the default functionality is completely specified by the user via *designer conversion functions*.

17.2.2 Supporting Applications During Change

Another important issue focuses on providing support for existing applications that depend on the old schema, when other applications change the shared schema according to their own requirements. Research to address this issue has followed along two possible directions, namely, views [RLR98, RR97, Ber92] and versions [SZ86, Lau97b]. These approaches rely on redundant information to continually support views and are examined mainly in the context of data-warehousing.

Query Language Extensions

In the project, Evolvable View Environment (EVE), Rundensteiner et al. utilize the redundancy in the information space to provide alternate sources for the same information. They propose an extension to SQL, called E-SQL

[RLN97a] which allows the user to express tolerance levels for the alternate information at the time of defining the query. This is particularly useful in a distributed or a federated environment where information about one class, say the `Student` class, is stored in more than one database and the user is aware of this replication. Thus, if we were to consider the given schema as being one in a large federation of schemas, then we could utilize the alternate sources for re-writing the queries. For example, if the user specifies that the class `Student` is equivalent to or some sub-set of it is equivalent to the class `Student'` in a different schema, then when the class `Student` is deleted by the schema evolution operation `delete-class`, we can do a query re-write using `Student'`. EVE offers many query language extensions for the distributed environment.

Another approach that uses query language extensions is AQL [Har94]. AQL provides extension to OQL to handle traversals by specifying a starting point and an ending point. However, this can handle only a limited set of schema evolution changes, particularly schema changes that alter the traversal paths. It cannot handle changes to traversal paths that are of length one. Similar proposal for path traversal has also been made by Kifer et al. [KKS92] in the language XSQL.

Hiding the Schema Change

Another approach that is perhaps more applicable in an environment where a replacement of information is not possible is the one proposed by Rundensteiner et al. in TSE - Transparent Schema Evolution [RR97]. TSE gives its users the impression that the requested schema evolution operation has

been performed, while in reality it only simulates it for the application or user that requested the change. This simulation of schema changes is done via a view mechanism which *hides* the change from the user. Other queries are unaffected as in their perception a change has not occurred. Using this approach queries do not need to be re-written. Similarly, a `add` change is propagated and applied to the base. Other queries defined on the same base are re-written to effectively project now less information than is available at the base.

A similar approach is taken by versioning [MS93, KC88, Lau96, Lau97a]. When a schema change is submitted, a new version of the schema is created. Older applications and queries can utilize the older schema version. Once again as in views, no re-write of the application or queries is required. However, versioning has its own set of problems that need to be resolved. Objects that are now created in the newer version need to be propagated backwards while objects created in the older version need to be propagated forward. Similar to views there can be a proliferation of versions and the system has to decide when it is possible to purge some versions.

17.2.3 XML Update Systems

The issue of change as well as that of propagating change from the source to the target has been looked at in other contexts. Schema evolution systems [Obj93, Cor00, SKC$^+$01, CJR98c] exist both commercially as well as in research projects and deal with structural changes in object database [Obj93] and relational databases [Cor00]. However, they do not deal with the propagation of change to targets that are derived from the now updated source.

In our previous work, we have developed *XEM* (XML Evolution Manager) [SKC+01], a taxonomy of schema evolution operations for a DTD and the corresponding set of XML documents. As part of this system we have also shown how the XML changes can be propagated to an object repository. Within XML there has been recent work done on propagation of XML updates to the underlying relational database system by Tatarinov et al. [TIHW01]. In their work, they describe a set of update methods for XML documents, present how existing XML query languages can be extended to support these update methods and finally show how these updates can be propagated to the persistent relational database system.

17.2.4 Other Related Work.

Rosenthal et al. [RS99a] have proposed a propagation framework that they have applied in the context of meta-data propagation in large multi-tier databases and also in the context of security information propagation through different levels of views. Our work is similar in that we apply the notion of propagation from the view to the base. However, most of our work is done with regards to the evolution of views.

17.3 Schema Integration and Data Transformation

There has been and continues to be much activity in the area of schema transformation and integration [HMN+99, MZ98, FK99, MIR93, CJR98c, RR87]. However, this work is typically specific to either the application domain or to a particular data model and does not deal with meta-modeling

[MR83, AT96, BR00, PR95, RR87]. Recent work related to ours are Clio [HMN$^+$99] a research project at IBM's Almaden Research Center and work by Milo and Zohar [MZ98]. Clio, a tool for creating mappings between two data representations semi-automatically (i.e., with user input) focuses on supporting querying of data in either the source or the target representation and on just in time cleansing and transformation of data. Milo et al. [MZ98] have looked at the problem of data translations based on schema-matching. They follow an approach similar to Atzeni et al.[AT96] and Papazoglou et al. [PR95] in that they define a set of translation rules to enable discovery of relationships between two application schemas. We can directly make use of translation algorithms from the literature, such as the algorithms for translating between an XML-DTD and relational schema [FK99] or mapping rules [MZ98], and represent them as cross algebra graphs. However, our focus is not discovering such algorithms for mapping but rather on the generic expressibility of any possible (future) mapping and its management. Work on equivalence of the translations between models [MIR93] is of particular importance as such properties of cross algebra graphs can also be established.

Meta-modeling has been utilized as a middle-ware medium to handle schema integration and data transformation over the last twenty years [MR83, AT96, BR00, PR95]. Papazoglou et al. [PR95] propose a middle-layer meta-model to accomplish transformations between the OO and relational data models. The transformations are accomplished by a set of predefined translation rules that can convert the OO or relational data models to and from the middle-layer meta-model. Using translations as basic

building blocks, they aim to automatically generate mappings from one given model to another at run-time. Atzeni et al. [AT96] have presented a framework to describe data models and application schemas. They focus on discovering translations between data models and hence application schemas. We make use of their graph model to express the data models and application schemas in our system. Commercially, Microsoft Repository [Ber99] and Rochade Information Model [Roc00] are meta-repositories that generically describe mainly data models for system integration purposes.

17.4 Data Models and Translations in the Middle Layer

In our work we have taken the mediator a la the middle layer approach. For any middle layer system, there are two important considerations (1) the data model, and (2) the mapping language. In our work, we have chosen to develop our own data model, the *Sangam* graph model, and based on that our set of algebra operators, namely the *cross algebra*. In this section we now give a comparison between the Sangam graph and the other choices for a data model; as well as a comparison of the cross algebra with other algebras, namely the XAlgebra [FFM+01a], an algebra for XQuery, and the relational algebra [FSS+97] that could potentially have been utilized for the middle-layer translation.

17.4.1 Schema Languages

An essential component of the middle-layer is to represent different schemas, i.e., various data structures and constraints. Therefore, in this comparison

we compare the most popular schema languages, XML-DTD [BPSM00], XML-Schema [Fal01], and relational schema [FSS+97] with the Sangam-graph. **XML-DTD** (DTD in short), a subset of SGML DTD, has limited capabilities compared to other schema languages. Its main building blocks consists of an *element* and an *attribute*. Data is represented by hierarchical element structures. The 2nd edition of the language specification became a W3C Recommendation on October 6, 2000. **XML-Schema**, according to [Fal01], is intended to be more expressive than DTD and can be expressed in XML notation. It has more advanced mechanisms beyond DTDs, such as inheritance for attributes and elements, user-defined data types, etc. XML-Schema became a W3C Recommendation on May 2nd, 2001. The specification is standard and favors its adoption by industry and research communities. **Relational schema** (language), perhaps the most widely used schema language, consists of two main building blocks, *relation* and *attribute*. Data is represented as flat relations. The relational model has been widely used since its standardization in mid 70s. **Sangam graph**, as developed by us is a modeling language, and for this reason represents the under-pinnings of the various schema languages. It can perhaps be seen as a simplification of the XML-schema [Fal01]. Its main building blocks are *nodes* and *edges*. It represents hierarchical structures as graphs.

To compare the different schema languages, Bonifati et al. [BL01] have developed a taxonomy. We use the categories defined as part of this taxonomy to now compare and evaluate the different schema languages as a modeling language for the middle-layer. For this evaluation, we use three main criteria based on the requirements for a middle-layer language as

layed out in Chapter 1: (1) the power of the schema language measured in terms of structural expressiveness, i.e., the ability of the language to express structural constructs from many different schema languages. To quantitatively measure this expressibility of schema languages, we look at the categories of (a) typing and extensibility. As languages support many different built-in domain types, it is essential that the middle layer language be able to model a superset of the built-in domain types in different languages; and (b) basic schema abstractions. This refers to the basic constructs that can be modeled by the schema language, for example, relation and attributes for the relational language, and elements, sub-elements and attributes for the XML schema. (2) the power of the schema language to express constraints. For this we look primarily at order management and constraints; and (3) the complexity of import/export utilities. When a schema language supports a feature fully or partially, denote this as a Yes or Partial. Otherwise we denote is as a No.

Criteria 1: Structural Expressive Power

Typing and Extensibility. Type-checking is an important feature of a schema language and can dramatically reduce the burden of application programs. Extensibility allows the type derivation from existing types. Data types can be *built-in* types or *user-defined* types. From a modeling language perspective, it is important that the language support a large set of built-in types, and also offer support for extending the built-in data types.

1. *Built-in* Type: A built-in type is a primitive provided by a schema language. Most schema languages support an array of built-in types including `string`, `integers` and in the case of XML based languages XML-related types such as `ID`, `NMTOKEN`. The numbers of such built-in types are:

   DTD: 10 XML-Schema: 45 Relational Schema: 32 Sangam-graph : not pre-set

   It is essential that a middle-layer language be able to model a large number of built-in domain types. In Sangam graphs, data types are modeled as data values for atomic nodes. Hence, there is no restriction imposed on the number of data types supported in Sangam graphs. By this criterion, Sangam graphs are better suited for the middle-layer than the other schema languages, with the XML-Schema being a close second.

2. *User-defined* Type: A capability to specify user-defined types greatly enhances the flexibility of the schema language. User-defined types in Sangam graphs are again modeled as data values for atomic nodes.

   DTD: `No` XML-Schema: `Yes` Relational Schema: `No`[1] Sangam graph : `No`

   By this criterion, both XML-schema and Sangam graphs are suitable

[1]In the extended relational model user-defined types can be specified.

as middle-layer languages.

3. *Type domain constraint*: The domain constraint of a type allows the definition of legal values admitted for that type. Some languages support a set of constructs to limit the valid domain values for data types. In the Sangam graph, type domain constraints such as range, precision and length can be modeled similar to the built-in types as data values for the atomic nodes. However, as modeling is our key emphasis, in our implementation, we currently do not provide any checking for these type domain constraints.

   DTD: `No` XML-Schema: `Yes` Relational Schema: `Yes` Sangam graph : `Yes`

   By this criterion, XML-Schema, relational schema and Sangam graph are all suitable as middle-layer languages.

Basic Schema Abstractions. In the above category, we focused primarily on the domain types offered by the different schema languages. We now focus on some of the basic schema constructs, and the features constraining the structure of a schema.

1. *Attribute default value*: When a value is not present in the XML document or relational table, a pre-determined default value is inserted. All schema languages support this feature. Sangam graph supports this via the atomic nodes.

DTD: `Yes` XML-Schema: `Yes` Relational Schema: `Yes` Sangam graph : `Yes`

2. *Optional vs. required attributes*: In all languages it is possible to express whether or not an attribute value is required or optional.

   DTD: `Yes` XML-Schema: `Yes` Relational Schema: `Yes` Sangam graph : `Yes`

3. *Choice*: Only one among several constructs (elements) are allowed.

   DTD: `Yes` XML-Schema: `Yes` Relational Schema: `No` Sangam graph : `No`

4. *Min and Max occurrences*: This feature describes whether a language can set up a participation/quantifier constraint for its content. For XML languages content includes sub-elements and attributes; for the relational language the content is simply the attributes; and for the Sangam graph it is all children nodes. Both DTD and XML-Schema allow min and max occurrences for sub-elements only, and not for the attributes. This is indicated by the word partial below.

   DTD: `Partial` XML-Schema: `Partial` Relational Schema: `Yes` Sangam graph : `Yes`

Criteria 2: Constraint Expressive Power

Order Management.

1. *Ordered list*: This feature investigates whether order in the content must be preserved. Again, for XML languages this content includes the sub-elements and the attributes, for relational language only the attributes, and for Sangam graph all children nodes. The XML languages support ordering of sub-elements only (and not of attributes) and hence are indicated as partial. In Sangam graph we make no distinction between the different children nodes whether they represent attributes or sub-elements. Hence, order can be assigned for all children nodes.

   DTD: `Partial` XML-Schema: `Partial` Relational Schema: `No` Sangam graph : `Yes`

2. *Unordered list*: This feature expresses the support of unorderness in the content.

   DTD: `No` XML-Schema: `Yes` Relational Schema: `Yes`[2] Sangam graph : `No`

Constraints.

1. *Uniqueness for attribute*: This defines the uniqueness for the attribute values. All languages support constructs that allow users to model

[2]This is the default for the relational language.

this uniqueness for attributes. Sangam graph allows for the modeling of uniqueness for attributes via the atomic nodes.

DTD: `Yes` XML-Schema: `Yes` Relational Schema: `Yes` Sangam graph : `Yes`

2. *Uniqueness for non-attribute*: XML languages like XML Schema allow specification of uniqueness not only for attributes but also for elements. For relational language, this is a non-issue. For the Sangam graph as there is no distinction between the different types of nodes (attribute or element node), hence similar uniqueness support exists for non-attributes.

   DTD: `No` XML-Schema: `Yes` Relational Schema: `N/A` Sangam graph : `Yes`

3. *Key*: In databases, being a key requires being both unique and not null. In Sangam graph, we model key constraints via atomic nodes as well.

   DTD: `Yes` XML-Schema: `Yes` Relational Schema: `Yes` Sangam graph : `Yes`

4. *Foreign key*: Foreign key describes both the attributes referencing keys and the attributes referenced by the key. In DTDs, partial support for this is provided via the ID and IDREF pairs.

DTD: `Partial` XML-Schema: `Yes` Relational Schema: `Yes`
Sangam graph : `Yes`

Criteria 3: Complexity of Import/Export Utilities

For a middle-layer data model, it is imperative that schema and data from other data models be represented by the middle layer language using simple default mappings. Dependent on the source schema language, the import or the export from the middle-layer can be easy or hard. For example, the import of hierarchical structures into a relational shema is hard. However, the export of the flat structure to a hierarchical structure is relatively easy. Thus, it is difficult to quantitatively measure the ease or difficulty of building these utilities for any middle-layer language.

Summary

From a technical standpoint it is not clear that one schema language (and hence the data model that it is bound to) is superior to all other schema languages, and hence a clear choice as the modeling language for the middle-layer. However, as can be seen by the above comparison, XML-schema and Sangam graph offer numerous advantages over the other two schema languages. For example, XML-schema offers explicit ordered and un-ordered capabilities (under the constraint expressive power criteria) for the content[3]. While Sangam graphs support ordered content, they do not provide any support for un-ordered content. On the other hand, Sangam graphs

[3] Content is as described above.

provide unlimited support for built-in types (under the structural expressive power criteria). While the built-in types supported by all the other languages are extensive, they are still limited. In terms of the complexity of import and export utilities, our third criteria, XML schema and Sangam graph offer the same level of capability.

Given its popularity and the push from industry, from a practical standpoint, XML-schema is clearly the best choice for a middle layer schema language. However, as it was made a recommendation in May 2001, it was not a valid choice at the start of this project. We believe that the Sangam graph is a simplification of the XML-schema, and hence much of the work presented is still valid if XML-Schema were the language of the middle-layer.

17.4.2 Query Algebras

Similar to the comparison for schema languages, we now compare three query algebras: XAlgebra [FFM$^+$01a], relational algebra [FSS$^+$97] and the cross algebra. The **XAlgebra** [FFM$^+$01a] is the new and upcoming algebra for the XML query. Its first draft appeared in early 2001, and there have been subsequent drafts in November 2001, and March 2002. The XAlgebra is also referred to as the XQuery formal semantics, post March 2002. This is currently a working draft. In the comparison here, we consider the XAlgebra as published in November 2001 [FFM$^+$01b]. The XAlgebra uses a simple type system that captures the essence of an XML-Schema and is based on the type system used in XDuce [HP00]. The **relational** algebra has been stable since the mid 70s, and is based on set theory. The relational algebra uses the relational type system. The **cross algebra** is the algebra

that we have developed. Its main emphasis is modeling of re-structuring in the middle layer as opposed to the full-fledged querying capabilities of many of the other algebras. The cross algebra uses the Sangam graph as its type system.

Our primary purpose is to compare the re-structuring capabilities of these algebras. To do this we have three main criteria: (1) the expressive power of the algebra operators. For this comparison, we have identified three algebra operators: projection, iteration, and join. These operators are the core re-structuring operators in any query language; (2) minimality of operators. We show this via examples; and (3) composition of expression. When the algebra supports a feature fully or partially, we denote it as a Yes or Partial. Otherwise we denote is as a No.

In the following discussions we give examples of the algebra operators in the three algebras. These examples are based on the schema and data given in Figures 17.1- 17.2. The schema in Figure 17.1 is shown in the type system of XAlgebra. The corresponding data is shown only in the type system of the XAlgebra.

Criteria 1: Algebra Expressiveness

Projection. The simplest operation is the projection. In the XAlgebra, this allows the extraction of a sub-tree from the given document. In the relational algebra, it allows for the extraction of a column from a set of columns. In the cross algebra there is no real equivalent of the projection operator. However, the semantics of the operations, i.e., the extraction of a particular node and its extent,can be represented by the a combination of the cross

```
type Bib =
  bib [ Book* ]
type Book =
  book [
    title  [ String ],
    year   [ Integer ],
    author [ String ]+
  ]
```

Figure 17.1: Schema in the XAlgebra Type System.

```
let bib0 : Bib =
  bib [
    book [
      title  [ "Data on
                 the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ],
    book [
      title  [ "XML Query" ],
      year   [ 2001 ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
    ]
```

Figure 17.2: Data shown in the Type System of XAlgebra.

and connect operators. Figures 17.3- 17.5 represent the projection operator in the three algebras XAlgebra, relational and cross algebra respectively. In the relational and the cross algebra, we simply try to represent the semantics of the algebra expression in Figure 17.3.

In XAlgebra (Figure 17.3), the expression has three components: algebra expression, result, and the type of the result. First is the algebra expression. Here `book0` is bound to a literal XML value, which is the data model representation of an XML document (shown in Figure 17.2). Second, following the ==>, is the value of the algebra expression. Lastly, following the ":", is the return type of the expression.

In the relational algebra (Figure 17.4), the projection is denoted by π. The subscript of π is the attribute to be projected. Within the brackets is the

```
book0/author
==> author [ "Abiteboul" ],
    author [ "Buneman" ],
    author [ "Suciu" ]
: author [ String ]+
```

Figure 17.3: Projection in XAlgebra.

$'\pi_{author}$(Book)

Figure 17.4: Projection in Relational Algebra.

$\otimes_{author'}$(author)

Figure 17.5: Projection in Cross Algebra.

relation name.

For the equivalent expression (Figure 17.3, the cross algebra expression shown in Figure 17.5, comprises of only one operator - the cross ($\otimes$) operator. The result of the expression is given as a subscript of the operator, and the input is given in the brackets.

Capability:

XAlgebra: `Yes` Relational: `Yes` Cross Algebra: `Yes`

Note that both the relational algebra and the cross algebra return all the `authors` for ***all*** the books. The projection in the XAlgebra will result in the `authors` only for the `book` to which `book0` is bound. To achieve the same result (as the relational and cross algebra) the equivalent XAlgebra expression is `bib0/book/author`. The relational equivalent for the XAlgebra expression `book0/author` is possible via a combination of the ***selection*** *(σ)*, ***projection*** π and ***join*** $\bowtie$ operators. However, there is no equivalent in cross algebra, as we do not have any support for conditionally selecting data, i.e., we do not support the selection operator.

Expressive Power:

XAlgebra: `High` Relational: `High` Cross Algebra: `Medium`

Iteration. Another common operation for XML documents is to iterate over elements in a document so that their content can be transformed into new content. While in XAlgebra this is an explicit construct, it is not explicitly supported in the relational or the cross algebra. Consequently, while some iteration expressions in XAlgebra have equivalent expressions in relational and cross algebra, there is no general equivalence. Figure 17.6 gives an example XAlgebra expression that performs iteration. Here each `book` is processed to list the `authors` before the `title`, and remove the year (and all other attributes). Figure 17.7 presents the equivalent cross algebra expression for the same. A relational equivalent for this can be given with a combination of selection, projection and join operators.

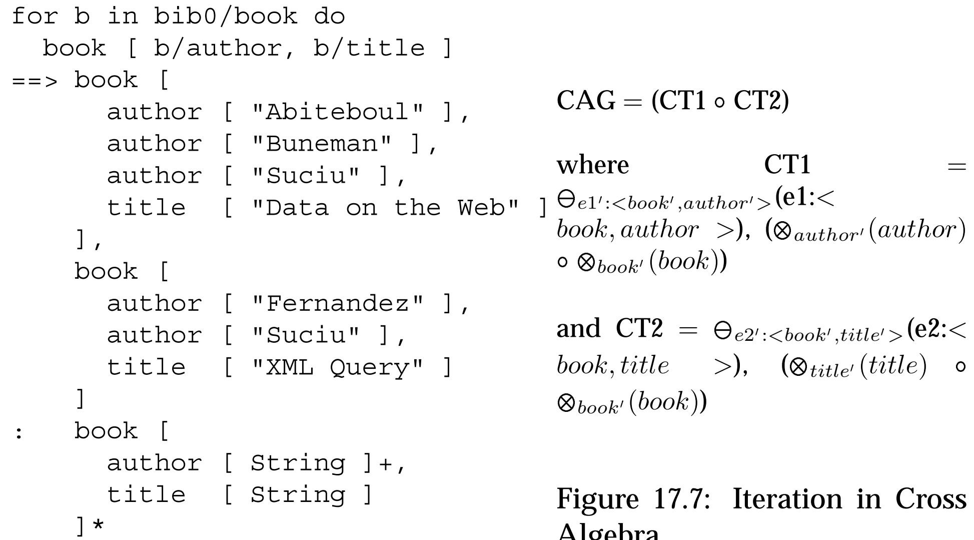
Capability:

XAlgebra: `Yes` Relational: `Yes` Cross Algebra: `Yes`

Again, note that the results in the XAlgebra give the `author` and `title` for *all* `books` in `bib0`; whereas the cross algebra expression results in the `authors` and `titles` for *all* `books` in *all* `bibs`. There is no equivalent expression in cross algebra that would provide this selection.

Expressive Power:

XAlgebra: `High` Relational: `High` Cross Algebra: `Medium`



```
for b in bib0/book do
  book [ b/author, b/title ]
==> book [
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ],
      title  [ "Data on the Web" ]
    ],
    book [
      author [ "Fernandez" ],
      author [ "Suciu" ],
      title  [ "XML Query" ]
    ]
:   book [
      author [ String ]+,
      title  [ String ]
    ]*
```

CAG = (CT1 ∘ CT2)

where CT1 = $\ominus_{e1':<book',author'>}$(e1:< $book, author$ >), ($\otimes_{author'}(author)$ ∘ $\otimes_{book'}(book)$)

and CT2 = $\ominus_{e2':<book',title'>}$(e2:< $book, title$ >), ($\otimes_{title'}(title)$ ∘ $\otimes_{book'}(book)$)

Figure 17.7: Iteration in Cross Algebra.

Figure 17.6: Iteration in XAlgebra.

Join. Another common operation is to join values from one or more documents. To illustrate joins, we give a second data source that defines book reviews. We only present the XAlgebra type system definitions here (Figure 17.8.).

Capability:

XAlgebra: `Yes` Relational: `Yes` Cross Algebra: `Partial`

Both XAlgebra and relational algebra explicitly support join, while in the cross algebra a limited form of the join can be accomplished with a combination of the other operators. The cross algebra Figure 17.9 gives an example XAlgebra expression that performs a *join* of the `reviews` and the `books`. Here nested `for` loops are used to join the two sources `review0`

```
type Reviews =
  reviews [
     book [
        title  [ String ],
        review [ String ]
     ]*
  ]
let review0 : Reviews =
    reviews [
       book [
          title  [ "XML Query" ],
          review [ "A darn fine book" ]
       ],
       book [
          title  [ \"Data on the Web\" ],
          review [ \"This is great!\" ]
       ]
    ]
```

Figure 17.8: Schema and Data for `Reviews` element in XAlgebra Type System.

```
for b in bib0/book do
  for r in review0/book do
     where value(b/title) = value(r/title) do
        book [ b/title, b/author, r/review ]
==> book [
      title  [ "Data on the Web" ]
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ],
      review [ "This is great!" ]
    ],
    book [
      title  [ "XML Query" ]
      author [ "Fernandez" ],
      author [ "Suciu" ],
      review [ "This is great!" ]
    ]
:   book [
      title  [ String ],
      author [ String ]+,
      review [ String ]
    ]*
```

Figure 17.9: Join in XAlgebra.

`review` $\bowtie_{review.title=book.title}$ `book`

Figure 17.10: Join in Relational Algebra.

and `book0`, where `review0` and `book0` are XML variables bound to `book` and `review` documents. The result combines the `title`, `authors`, `reviews` for `book0`. Figure 17.10 gives an equivalent relational algebra. However, a join cannot be accomplished using the cross algebra operators. Only a Cartesian product can be expressed using the cross algebra operators. This Cartesian product is given in Figure 17.11.

Expressive Power:

XAlgebra: `High`   Relational: `High`   Cross Algebra: `Medium`

CAG = CT1 ∘ CT2 ∘ CT3 ∘ CT4

where CT1 = $\ominus_{e1':<book',author'>}$(e1:$<$ $book, author$ $>$), ($\otimes_{author'}(author)$ ∘ $\otimes_{book'}(book)$)

CT2 = $\ominus_{e2':<book',title'>}$(e2:$<$ $book, title$ $>$), ($\otimes_{title'}(title)$ ∘ $\otimes_{book'}(book)$)

CT3 = $\ominus_{e3':<book',reviews.title'>}$(e3:$<$ $reviews.book, reviews.title$ $>$), ($\otimes_{review.title'}(reviews.title)$ ∘ $\otimes_{book'}(reviews.book)$)

CT4 = $\ominus_{e4':<book',reviews.review'>}$(e4:$<$ $reviews.book, reviews.review$ $>$), ($\otimes_{reviews.review'}(reviews.review)$ ∘ $\otimes_{book'}(reviews.book)$)

Figure 17.11: Join in Cross Algebra.

Other Operators. In addition to the algebra operators that we have presented here, the XAlgebra also provides the following operators: selection, quantification, grouping, aggregation, functions and structural recursion. The operators selection, quantification, aggregation and grouping have an equivalent relational algebra expression, albeit on the relational model that does not have the XML supported nested structure. These operators do not have an equivalent in the cross algebra. Conversely, the cross algebra includes the ***subdivide*** operator which does not have an equivalent in either the XAlgebra or the relational algebra. However, we believe the XAlgebra can be easily extended to include the subdivide operator. Another observation that we have made is that a ***select*** operator would greatly enhance the re-structuring capabilities of the cross algebra. This is we believe a simple extension of the cross algebra.

Criteria 2: Minimality of Operators

In this category, we explore the minimality of the re-structuring algebra operators. As can be seen by the examples given in Figures 17.3, 17.6, 17.9, there are no XAlgebra operators that provide "minimal semantics". For example, the simplest operation in XAlgebra, *projection* is a complex operation that can be broken down into a selection and projection in the relational algebra. We believe that a minimal semantics algebra operation set has not been identified for the XAlgebra. On the other hand, for both the relational and the cross algebra, the operators are defined to have minimal semantics.

Minimality of Operators:

XAlgebra: `Low` Relational: `High` Cross Algebra: `High`

Criteria 3: Composition Capabilities

A key component for all algebras is the composition of larger expressions based on the set of algebra operators. All algebras provide this composition to varying degrees. XAlgebra provides nesting of expressions. Similarly, relational algebra and the cross algebra also provide nesting of their operators. In addition, the cross algebra provides a dependency composition, context dependency, that allows a grouping of operators that have some execution dependency and the outputs of all of which are represented in the final output. Their is no equivalent composition in the relational algebra and the XAlgebra.

Composition Capabilities:

XAlgebra: `Medium` Relational: `Medium` Cross Algebra: `High`

Summary

To summarize, from a technical standpoint, we believe that for re-structuring in the middle-layer, there are no clear cut winners. Each algebra has its advantages and disadvantages. In terms of re-structuring expressive power and minimality of operators, the relational algebra is the most complete and stable. Similar to the relational algebra, the cross algebra offers a powerful and complete[4] set of operators for transformations and re-structuring. Moreover, the cross algebra operators are minimal and provide easy composition of larger expressions for complete schema transformations. However, it would be greatly enhanced if we were to add the *select* operator. In contrast, while the XAlgebra is powerful in terms of expressive power, it falls short on the minimality.

From a practical standpoint, however, the cross algebra lacks the querying capabilities provided by the XAlgebra and the relational algebra. One of our conclusions from Section 17.4.1 is that XML-Schema is probably the best practical choice for the middle-layer data model. Based on this, we would concede that the use of the XAlgebra as the middle layer translation system may perhaps be the most practical choice as well. However, we believe that the re-structuring operators of XAlgebra need to be re-worked

[4]While we do not show the completeness of the cross algebra, this result follows from similar results in the graph linear transformation theory [GY98].

with the addition of the subdivide operator, and with minimality in mind; and perhaps a marriage of the cross-algebra and the XAlgebra is worthwhile exploration for a future middle-layer mapping algebra.

In conclusion here, we would like to re-iterate that the cross algebra was designed to provide modeling of transformation in the middle-layer. Towards that end we have provided a set of modeling primitives, the cross algebra, the semantics associated with each operator and one explicit mechanism to execute these operators. The cross algebra is based on the graph linear transformation theory, which has shown that these operations are complete with respect to linear transformations [GY98]. With the above comparison we have shown that many of these cross algebra expressions can be easily expressed in current query algebras such as the relational and the XAlgebra. Thus, for execution purposes the cross algebra expressions can be expressed as relational or XAlgebra expressions.

Part IV

Conclusions and Future Work

Chapter 18

Conclusions and Future Work

A key aspect of any persistent information is the simple fact that it changes. Managing change, which includes the specification and optimization of change as well the maintenance of any derived information, has been an important area of research. Today, for every data model there exists a well-defined set of change primitives [BKKK87, FFM$^+$95, SKC$^+$01] that can alter both the structure (the schema) and the data. Several proposals also exist for incrementally propagating this primitive change to any derived information (or view) [ZGMHW95, GB95, RLN97a]. However, this existing support is lacking in two ways. First, change primitives as presented in literature are very limiting in terms of their capabilities allowing users to simply add or remove schema elements. More complex types of changes such the merging or splitting of schema elements are not not supported in a principled manner. Secondly, algorithms for maintaining derived information often do not account for the potential heterogeneity between the source and the target. The goal of this dissertation is to provide solutions

that address these two key issues.

18.1 Contributions of this Dissertation

18.1.1 Support for Complex Changes

Support for complex changes in databases exists in the form of pre-defined set of change operations that can be invoked for different parameters [Bré96, Ler00]. This work [Bré96, Ler00] defines a set of high-level primitives such as *merge*, *split* and *inline* for object-oriented databases, in particular for O_2. However, it is difficult to a-priori define (1) all possible complex operations; and (2) all the possible semantics for the set of complex operations. For example, a *merge* of two classes can be accomplished by combining the attributes and the extents of the two classes, or by forming a new class that contains only the common set of attributes for the two classes. For any change beyond the pre-defined set, a user must therefore write programs that allow him or her to manipulate the structure of their database as desired. Such an approach is error-prone, provides no guarantees for consistency of the database, does not lend itself to any kind of verification or optimization, and is not portable from one database to another.

In Part II of this dissertation, we have have identified the fundamental components of any complex change - the change expressed in terms of a set of primitive changes and the corresponding, potentially complex, data changes. Based on this hypothesis, we have developed SERF [CJR98c] - an extensible and re-usable framework for schema evolution. The framework is extensible in that the user is not limited to a fixed set of change

primitives. Instead, a user can define their own set of schema changes using a combination of schema evolution primitives and query language, in our case OQL (Object Query Language). Any complex change defined by the user can be stored in the system (SERF system) as a *template* for similar changes by other users. We have developed a prototype of the SERF system, OQL-SERF, that is defined as a thin-layer on top of existing OODB systems. We have utilized PSE (Persistent Storage Engine) [O'B97] as our OODB system.

As noted earlier one of the key advantages of such a framework over ad-hoc user programs, is the system's guarantee for consistency of database post-execution of the complex change. To facilitate this, we consider two types of consistency - user-level and system-level. A system-level consistency guarantees that all invariants of the data model will be preserved after the execution of a complex change. We have formally shown that this system-level consistency is indeed guaranteed by the SERF system [CRed]. A user-level consistency guarantees that a user contract is satisfied after the execution of a change. To facilitate this, we have developed the notion of contracts, a set of pre- and post- conditions, that can be defined for a SERF template [CRH01]. The pre-conditions are verified prior to the execution of the template and the post-conditions are validated after the template execution. The template is *aborted* (transaction is rolled back) if the post-conditions are not met.

Another area that we have explored in the context of this work is the optimization of schema evolution operations. Clearly, schema evolution is a very expensive operation and this expense is exaggerated by the now

more complex changes. We have pursued two approaches for optimizing this expense: (1) by the reduction of the schema evolution operations based on some heuristics [CRH00a]; and (2) by reducing the opportunity for roll-backs or aborted templates [CRH01]. In this dissertation, we have presented both of these optimization techniques in Chapters 6.1 and 5.3.

To validate the SERF framework and the main concepts behind it, we have represented numerous complex changes presented in literature as SERF templates and have validated them using our prototype system [RCL+99, CJR98c, CJR98a]. Moreover, we have extended the basic concepts of the SERF framework to also address complex schema evolution operations that involve relationships between two or more classes [CRH00b]. We have further validated the SERF framework by its application to Web re-structuring [CCR00, RCC00].

18.1.2 Maintenance of Heterogeneous Views

Views and maintenance of views with respect to source changes has been a lively topic in database research [ZGMHW95, GB95]. Much of the work in this area has been based on the assumption that both the source and the view are defined within the confines of the same data model, and often within the same database. However, today this world is fast changing. Today many application engineers struggle to not only publish their relational, object or ASCII file data on the Web but to also integrate information from diverse sources. However, current information integration technology has moved away from traditional algebra-based views to static a-priori algorithms. Consequently, maintenance in such scenarios is also

limited to static a-priori algorithms [TIHW01] that rely on the underlying translation algorithm between the two data models, in this case XML and relational data model. Such an approach is not extensible to any new translation technique, a new view, between the same two data models, much less a different pair of data models. To meet the needs of information integration and its subsequent maintenance for the future, we consider current technology in-adequate.

In Part III of this dissertation, we address this problem using a two-pronged approach. We first define an algebra, a cross algebra, that can define views such that there is no restriction that forces the view and the source data models to be the same. To accomplish this, we have defined the cross algebra that covers the class of linear transformations [GY98] applied to a graph (Sangam graph) that represents schemas from different data models. To enable the translation of an entire schema in one data model to a schema in another data model, we also allow the composition of these algebra operators by the traditional *derivation* composition as well as by our *context dependency composition*. In the context of this work, we present the evaluation algorithm for executing a cross algebra expression and we show that the evaluation algorithm (1) terminates, and (2) it produces a valid output.

To validate our proposed ideas of cross algebra we have implemented a prototype system and conducted several experiments that (1) validate in practice that we are indeed able to express a variety of translation algorithms. We have tested this on the different translation algorithms found in literature [ZLMR01, FK99, SHT^{+}99, CFLM00, SYU99]; and (2) give a mea-

sure of the performance of the prototype system. In this dissertation, we report these results. All our tests were conducted on the relational and the XML data models.

Based on the cross algebra operators (and graphs), we have also developed an incremental update propagation algorithm, *Gen_Propagation* , that allows us to propagate a change, schema or data change, from the source to the target. We have shown that this incremental propagation algorithm produces the same result as complete re-computation. Furthermore, we have experimentally measured the performance gains that can be achieved using the incremental propagation algorithm. A key advantage to this algorithm is that it is not tied to any particular translation algorithm between two data models nor is it tied to any two particular data models. To validate this we have applied this algorithm to propagate changes from XML to relational database, where the XML documents have been mapped to the relational data using (a) basic inlining [STZ+99]; (b) shared inlining [STZ+99]; and (c) ident. In this dissertation, we report on this algorithm and on our results, both formal and experimental.

18.2 Future Directions

18.2.1 Optimization of Schema Evolution

As noted earlier, schema evolution is a very expensive process. To enable dynamic or on-line evolution we must try to reduce the down-time/ unavailability of the system. Researchers have looked at improving system availability during schema evolution by proposing execution strate-

gies such as deferred execution [Tec94, FMZ94b]. Kahler et al. [KR87] have looked at pre-execution optimization for reducing the number of update messages that are sent to maintain replicated sites in the context of distributed databases. We take a similar approach (merge, cancel, eliminate) for optimizing a sequence of schema evolution operations. To the best of our knowledge ours is the first effort to provide an optimization strategy for a sequence of changes prior to execution a la Kahler [KR87]. However, while applying this work in the context of SERF templates we nevertheless limit our work to dealing with pure schema evolution sequences. That is, we do not consider the presence of OQL queries which are an essential component of SERF templates. The heuristics presented here must be re-examined now in the presence of OQL queries. One idea would be to combine traditional query optimization techniques with the heuristics for reducing the schema evolution sequences to enable a more *global optimization* of SERF templates.

Another venue for optimization that we have not examined is the use of the optimization heuristics together with the lazy or deferred update techniques proposed by Zicari et al. [FMZ94b]. The deferred update approach applies updates during query evaluation. That is, when a data set is requested, if there are any updates applicable to the data set, then the update is applied before presenting the data set to the user. With a large data set and a large number of updates, this can potentially cause poor query response times. A possible optimization here could be the utilization of our optimization heuristics to reduce the number of evolution operations/updates that are applied to the data set.

18.2.2 Integrating the Cross Algebra With Existing Local Algebra

One of the limitations of the cross algebra is that it is currently deployable only as a middle-layer algebra. That is, it is a stand-alone algebra and is not integrated with any existing algebras such as the relational algebra or XML algebra. A very interesting future direction would be incorporate the cross algebra as an extension to SQL and hence the relational algebra. The binding of the cross algebra to a data model, XML for example, would enable relational results to be represented as XML documents. Conversely, incorporating the cross algebra[1] with the XML algebra would enable relational outputs to be produced by the querying of XML documents.

The potential gains of such an approach are immense. Traditional query optimization (for relational or XML for example) can be extended to now incorporate the cross algebra operators. For example, to produce a *sorted* relational output from XML documents, it may be more efficient to pipeline the cross algebra operators and the *sort* operator on the XML documents, rather than first sorting the XML data and then translating it to relational tables. Query translation techniques can also be made more seamless with such an integration of the cross algebra and the local algebra (XML or relational).

A key assumption for our propagation algorithm is the fact that a change is made to the source (relational or XML) and then propagated to the target (relational or XML). We do not consider the fact that the relational source for example may itself be a view defined over multiple relational sources.

[1] If the algebra was bound to the relational model.

In such a scenario, a more efficient propagation algorithm would incorporate existing propagation algorithms for relational views with the propagation algorithm, *Gen_Propagation* that we have developed here. Such an incorporation is possible only if we are able to integrate the cross algebra with the local algebra.

Finally, another possible extension to this dissertation is the merging of the capabilities presented in Parts II and III. In Part II we have focused on specifying complex change within one data model. In Part III we focus our efforts on propagating simple changes across data model boundaries. A possible extension of our work is now possibly propagating complex changes, a la SERF, through the cross algebra. One solution to this could lie in the simplification of the problem achieved by perhaps integrating the cross and the local algebra.

18.2.3 Inverse Update Propagation

In large enterprise systems, information is often dispersed over multiple-tiers in a combination of physical (source) and virtual (view) databases in an effort to service a large community of users. Design systems are an example of large-scale systems that have to service the needs of many users, often hundreds of users [PMD95]. In [PMD95], MacKellar and Peckham describe how a large-scale design is decomposed into a number of specialized tasks each requiring its own representation of the design. Users often specialize in one aspect of the design and thus only deal with one representation of the design, also termed a perspective or a view. In such large often multi-tier systems, views (virtual databases) are built upon either another

tier of derived information or directly upon the source database systems. From a user's perspective there is no distinction between the view that they work on and the source(s) in the sense of the services and support they expect for the system. To provide this transparency to the user we need to effectively support views as *first-class citizens* [RS99a] and handle execution of all types of changes seamlessly. Today most database systems provide support for views but limit them to *read-only* access or at best allow data updates [RS99b]. This problem is further amplified when you take into consideration the fact that the tiers may potentially exist in different data models.

A possible extension of this dissertation is to now enable inverse propagation of changes, i.e., propagate changes from the view (target) to the source. A first step to this work could be the propagation of simple change within the cross algebra with potential extensions to an integrated algebra (combination of cross and local algebra), and more complex changes.

Appendix A

DTDs Used in Experiments

Figure A.1 gives the complete `auction.dtd`. Figures A.5 and A.6 present the `personal.dtd` and `play.dtd` respectively.

```
<!-- DTD for auction database -->
<!-- $Id: auction.dtd,v 1.15 2001/01/29
 21:42:35 albrecht Exp $ -->

<!ELEMENT site            (regions, categories,
                           catgraph, people,
                           open_auctions,
                           closed_auctions)>
<!ELEMENT categories      (category+)>
<!ELEMENT category        (name, description)>
<!ATTLIST category        id ID #REQUIRED>
<!ELEMENT name            (#PCDATA)>
<!ELEMENT description     (text | parlist)>
<!ELEMENT text            (#PCDATA | bold
                           | keyword | emph)*>
<!ELEMENT bold            (#PCDATA | bold
                           | keyword | emph)*>
<!ELEMENT keyword         (#PCDATA | bold
                           | keyword | emph)*>
<!ELEMENT emph            (#PCDATA | bold
                           | keyword | emph)*>
<!ELEMENT parlist         (listitem)*>
<!ELEMENT listitem        (text | parlist)*>
<!ELEMENT catgraph        (edge*)>
<!ELEMENT edge            EMPTY>
<!ATTLIST edge            from IDREF #REQUIRED to
                          IDREF #REQUIRED>
```

Figure A.1: The `auction.dtd` of the Xmark Benchmark [SWK+01].

```
<!ELEMENT regions         (africa, asia,
                           australia, europe,
                           namerica, samerica)>
<!ELEMENT africa          (item*)>
<!ELEMENT asia            (item*)>
<!ELEMENT australia       (item*)>
<!ELEMENT namerica        (item*)>
<!ELEMENT samerica        (item*)>
<!ELEMENT europe          (item*)>
<!ELEMENT item            (location, quantity,
                           name, payment,
                           description, shipping,
                           incategory+, mailbox)>
<!ATTLIST item            id ID #REQUIRED
                          featured CDATA #IMPLIED>
<!ELEMENT location        (#PCDATA)>
<!ELEMENT quantity        (#PCDATA)>
<!ELEMENT payment         (#PCDATA)>
<!ELEMENT shipping        (#PCDATA)>
<!ELEMENT reserve         (#PCDATA)>
<!ELEMENT incategory      EMPTY>
<!ATTLIST incategory      category IDREF #REQUIRED>
<!ELEMENT mailbox         (mail*)>
<!ELEMENT mail            (from, to, date, text)>
<!ELEMENT from            (#PCDATA)>
<!ELEMENT to              (#PCDATA)>
<!ELEMENT date            (#PCDATA)>
<!ELEMENT itemref         EMPTY>
<!ATTLIST itemref         item IDREF #REQUIRED>
<!ELEMENT personref       EMPTY>
<!ATTLIST personref       person IDREF #REQUIRED>
<!ELEMENT people          (person*)>
```

Figure A.2: The `auction.dtd` of the Xmark Benchmark [SWK+01].

```
<!ELEMENT person          (name, emailaddress,
                           phone?, address?,
                           homepage?, creditcard?,
                           profile?, watches?)>
<!ATTLIST person          id ID #REQUIRED>
<!ELEMENT emailaddress    (#PCDATA)>
<!ELEMENT phone           (#PCDATA)>
<!ELEMENT address         (street, city, country,
                           province?, zipcode)>
<!ELEMENT street          (#PCDATA)>
<!ELEMENT city            (#PCDATA)>
<!ELEMENT province        (#PCDATA)>
<!ELEMENT zipcode         (#PCDATA)>
<!ELEMENT country         (#PCDATA)>
<!ELEMENT homepage        (#PCDATA)>
<!ELEMENT creditcard      (#PCDATA)>
<!ELEMENT profile         (interest*, education?,
                           gender?,
                           business, age?)>
<!ATTLIST profile         income CDATA #IMPLIED>
<!ELEMENT interest        EMPTY>
<!ATTLIST interest        category IDREF #REQUIRED>
<!ELEMENT education       (#PCDATA)>
<!ELEMENT income          (#PCDATA)>
<!ELEMENT gender          (#PCDATA)>
<!ELEMENT business        (#PCDATA)>
<!ELEMENT age             (#PCDATA)>
<!ELEMENT watches         (watch*)>
<!ELEMENT watch           EMPTY>
<!ATTLIST watch           open_auction IDREF #REQUIRED>
<!ELEMENT open_auctions   (open_auction*)>
```

Figure A.3: The `auction.dtd` of the Xmark Benchmark [SWK+01].

```
<!ELEMENT open_auction    (initial, reserve?,
                           bidder*, current,
                           privacy?, itemref,
                           seller, annotation,
                           quantity, type, interval)>
<!ATTLIST open_auction    id ID #REQUIRED>
<!ELEMENT privacy         (#PCDATA)>
<!ELEMENT initial         (#PCDATA)>
<!ELEMENT bidder          (date, time,
                           personref, increase)>
<!ELEMENT seller          EMPTY>
<!ATTLIST seller          person IDREF #REQUIRED>
<!ELEMENT current         (#PCDATA)>
<!ELEMENT increase        (#PCDATA)>
<!ELEMENT type            (#PCDATA)>
<!ELEMENT interval        (start, end)>
<!ELEMENT start           (#PCDATA)>
<!ELEMENT end             (#PCDATA)>
<!ELEMENT time            (#PCDATA)>
<!ELEMENT status          (#PCDATA)>
<!ELEMENT amount          (#PCDATA)>
<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction  (seller, buyer, itemref,
                           price, date, quantity,
                           type, annotation?)>
<!ELEMENT buyer           EMPTY>
<!ATTLIST buyer           person IDREF #REQUIRED>
<!ELEMENT price           (#PCDATA)>
<!ELEMENT annotation      (author, description?,
                           happiness)>

<!ELEMENT author          EMPTY>
<!ATTLIST author          person IDREF #REQUIRED>
<!ELEMENT happiness       (#PCDATA)>
                           personref, increase)>
```

Figure A.4: The `auction.dtd` of the Xmark Benchmark [SWK+01].

```
<?xml encoding="UTF-8"?>
<!ELEMENT personnel (person)+>

<!ELEMENT person (name,email*,url*,link?)>
<!ATTLIST person id ID #REQUIRED>
<!ATTLIST person note CDATA #IMPLIED>
<!ATTLIST person salary CDATA #IMPLIED>

<!ELEMENT name (family,given)>

<!ELEMENT family (#PCDATA)>

<!ELEMENT given (#PCDATA)>

<!ELEMENT email (#PCDATA)>

<!ELEMENT url EMPTY>
<!ATTLIST url href CDATA 'http://'>

<!ELEMENT link EMPTY>
<!ATTLIST link manager IDREF #IMPLIED>
<!ATTLIST link subordinates IDREFS #IMPLIED>

<!NOTATION gif PUBLIC '-//APP/Photoshop/4.0'
 'photoshop.exe'>
```

Figure A.5: The `personal.dtd` used for Sangam testing.

```
<!-- DTD for Shakespeare    J. Bosak    1994.03.01,
  1997.01.02 -->
<!-- Revised for case sensitivity 1997.09.10 -->
<!-- Revised for XML 1.0 conformity 1998.01.27
 (thanks to Eve Maler) -->

<!-- <!ENTITY amp "&#38;#38;"> -->
<!ELEMENT PLAY     (TITLE, FM, PERSONAE, SCNDESCR,
                    PLAYSUBT, INDUCT?, PROLOGUE?,
                    ACT+, EPILOGUE?)>
<!ELEMENT TITLE    (#PCDATA)>
<!ELEMENT FM       (P+)>
<!ELEMENT P        (#PCDATA)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
<!ELEMENT PGROUP   (PERSONA+, GRPDESCR)>
<!ELEMENT PERSONA  (#PCDATA)>
<!ELEMENT GRPDESCR (#PCDATA)>
<!ELEMENT SCNDESCR (#PCDATA)>
<!ELEMENT PLAYSUBT (#PCDATA)>
<!ELEMENT INDUCT   (TITLE, SUBTITLE*,
                   (SCENE+|(SPEECH|STAGEDIR|SUBHEAD)+))>
<!ELEMENT ACT      (TITLE, SUBTITLE*, PROLOGUE?,
                    SCENE+, EPILOGUE?)>
<!ELEMENT SCENE    (TITLE, SUBTITLE*,
                   (SPEECH | STAGEDIR | SUBHEAD)+)>
<!ELEMENT PROLOGUE (TITLE, SUBTITLE*, (STAGEDIR
                                      | SPEECH)+)>
<!ELEMENT EPILOGUE (TITLE, SUBTITLE*, (STAGEDIR
                                      | SPEECH)+)>
<!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR
                                      | SUBHEAD)+)>
<!ELEMENT SPEAKER  (#PCDATA)>
<!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ELEMENT SUBHEAD  (#PCDATA)>
```

Figure A.6: The `play.dtd` available with JAXP1.1.

Bibliography

[AH90] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *Readings in Object-Oriented Database Systems*, pages 186–196, 1990.

[ANS92] ANSI Standard. The SQL 92 Standard. http://www.ansi.org/, 1992.

[AT96] P. Atzeni and R. Torlone. Management of Multiple Models in an Extensible Database Design Tool. In Peter M. G. Apers and et al., editors, *Advances in Database Technology - EDBT'96, Avignon, France, March 25-29*, LNCS. Springer, 1996.

[AYBS97] S. Amer-Yahia, P. Bréche, and C.S.D Santos. Object Views and Updates. In *ISI*, 1997.

[BB99] P.A. Bernstein and T. Bergstraesser. Meta-Data Support for Data Transformations Using Microsoft Repository. In *IEEE Bulletin - Special Issue on Database Tranformation Technology*, pages 9–14, 1999.

[BCGMG97] E. Bertino, B. Catania, J. Garcia-Molina, and G. Guerrini. A Formal Model of Views for Object-Oriented Database Systems. *Journal of Theory and Practice of Object Systems (TAPOS)*, 1997.

[BCVG86] A. Buchmann, R. Carrera, and M. Vazquez-Galindo. A Generalized Constraint and Exception Handler for Object-Oriented CAD-DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 38–49, 1986.

[Ber92] E. Bertino. A View Mechanism for Object-Oriented Databases. In *3rd Int. Conference on Extending Database Technology*, pages 136–151, March 1992.

[Ber99] Bernstein, P. A. and Bergstraesser, T. et al. Microsoft Repository Version 2 and the Open Information Model. *Information Systems*, 2(24):71–98, 1999.

[BFK95] P. Breche, F. Ferrandina, and M. Kuklok. Simulation of Schema Change using Views. In *Int. Conference and Workshop on Database and Expert Systems Applications*, 1995.

[BH93] P.L. Bergstein and W.L Hursch. Maintaining Behavioral Consistency during Schema Evolution. In *International Symposium on Object Technologies for Advanced Software*, pages 176–193, 1993.

[BHJ+96] P.J. Black, K.M. Hall, M.D. Jones, T.N. Larson, and P.J. Windley. A Brief Introduction to Formal Methods. In *Proceedings of CICC*, pages 377–380, 1996.

[BK90] F. Bancilhon and W. Kim. Object-Oriented Database Systems: In Transition. In *SIGMOD RECORD*, volume 19, December 90.

[BKKK87] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD*, pages 311–322, 1987.

[BL01] A. Bonifati and D. Lee. Comparative Examples of XML Schema and Query Languages. In *Internet Document*, January 2001.

[Bla98] P.E. Black. *Axiomatic Semantic Verification of a Secure Web Server*. PhD thesis, Brigham Young University, February 1998.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methedologies for Database Schema Integration. *ACM Computing Surveys*, 18(4), December 1986.

[BMO+89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Man-

agement System. In *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, 1989.

[Boo94] G. Booch. *Object-Oriented Analysis and Design.* Benjamin Cummings Pub., 1994.

[BPSM00] T. Bray, J. Paoli, and C.M. (Eds). Sperberg-McQueen. Extensible Markup Language (XML)1.0. http://www.w3c.org/TR/2000/REC-xml-20001006, October 2000.

[BR00] P. A. Bernstein and E. Rahm. Data Warehouse Scenarios for Model Management. In *International Conference on Conceptual Modeling*, 2000.

[Bré96] P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.

[BS81] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.

[CCR00] L. Chen, K.T. Claypool, and E.A Rundensteiner. SERFing the Web: The Re-Web Approach for Web Re-Structuring. *World Wide Web Journal: Special Issue on World Wide Web Data Management*, 2000. Final Revision Submitted Feb. 2000.

[Cea97] R.G.G Cattell and et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., 1997.

[CFLM00] B. Catania, E. Ferrari, A. Levy, and A. Meldelzon. XML and Object Technology. In *ECOOP Workshop on XML and Object Technology, LNCS 1964*, pages 191–202, 2000.

[CJR98a] K.T. Claypool, J. Jin, and E.A. Rundensteiner. OQL_SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In *Centre for Advanced Studies Conference*, pages 108–122, November 1998.

[CJR98b] K.T. Claypool, J. Jin, and E.A. Rundensteiner. OQL_SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. Technical Report WPI-CS-TR-98-14, Worcester Polytechnic Institute, July 1998.

[CJR98c] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, pages 314–321, November 1998.

[CJR98d] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF:Schema Evolution through an Extensible, Re-usable and Flexible Framework. Technical Report WPI-CS-TR-98-9, Worcester Polytechnic Institute, May 1998.

[Cla92] S.M. Clamen. Type Evolution and Instance Adaptation. Technical Report CMU-CS-92-133R, Carnegie Mellon University, School of Computer Science, 1992.

[Clu98] S. Cluet. Designing OQL: Allowing objects to be queried. *Journal of Information Systems*, 23(5):279–305, 1998.

[CNR99] K.T. Claypool, C. Natarajan, and E.A. Rundensteiner. Optimizing the Performance of Schema Evolution Sequences. Technical Report WPI-CS-TR-99-06, Worcester Polytechnic Institute, February 1999.

[CNR00] K.T. Claypool, C. Natarajan, and E.A. Rundensteiner. Optimizing the Performance of Schema Evolution Sequences. In *ECOOP Symposioum on Object Databases*, 2000.

[Cor00] Oracle Corporation. Oracle 8i. http://www.oracle.com/ip/deploy/database/8i/index.html, 2000.

[CRCK98] K. Claypool, E.A. Rundensteiner, L. Chen, and B. Kothari. Re-usable ODMG-based Templates for Web View Generation and Restructuring. In *WIDM'98*, pages 314–321, 1998.

[CRH00a] K. T. Claypool, E. A. Rundensteiner, and G.T. Heineman. Evolving the Software of a Schema Evolution System. In *Ninth International Workshop on Foundations of Models and Languages for Data and Objects*, September 2000.

[CRH00b] K. T. Claypool, E. A. Rundensteiner, and G.T. Heineman. ROVER: A Framework for the Evolution of Relationships. In *Nineteenth International Conference on Conceptual Modeling*, October 2000.

[CRH01] K.T. Claypool, E.A Rundensteiner, and G.T. Heineman. ROVER: Flexible Yet Consistent Evolution of Relationships. *Data and Knowledge Engineering*, 39(1):27–50, Oct 2001.

[CRed] K.T. Claypool and E.A. Rundensteiner. SERF:Schema Evolution through an Extensible, Re-usable and Flexible Framework. , submitted.

[CRZ+01] K.T. Claypool, E.A. Rundensteiner, X. Zhang, H. Su, H. Kuno, W-C. Lee, and G. Mitchell. SANGAM: A Solution to Support Multiple Data Models, Their Mappings and Maintenance. In *Demo Session Proceedings of SIGMOD'01*, 2001.

[Day89] U. Dayal. Queries and views in an object-oriented data model. In *Intern. Work. on Data Base Programming Language 2*, 1989.

[DB78] U. Dayal and P. Berstein. On the Updatability of Relational Views. In *VLDB*, 1978.

[DK97] S.B. Davidson and A.S. Kosky. WOL: A Language for Database Transformations and Constraints. In *IEEE Int. Conf. on Data Engineering*, pages 55–65, 1997.

[DKE94] S.B. Davidson, A.S. Kosky, and B. Eckman. Facilitating Transformations in a Human Genome Project Database. In *Int. Conf. on Information and Knowledge Management*, pages 423–432, 1994.

[Dmi98] M. Dmitriev. The First Experience of Class Evolution Support in PJama. In *Third International Conference on Persistence and Java*, 1998.

[DZ91] C. Delcourt and R. Zicari. The Design of an Integrity Consistency Checker (ICC) for an Object Oriented-Database System. In P. America, editor, *ECOOP*, pages 97–117, 1991.

[EN96] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 1996.

[Fal01] D.C. Fallside. XML Schema 1.1. http://www.w3c.org/TR/xmlschema-0, May 2001.

[FFHI72] J.P. Fry, R.L. Frank, and E.A. Hershey III. A Development Model for Data Translation. In A. L. Dean., editor, *Proceedings of 1972 ACM-SIGFIDET Workshop on Data Description, Access and Control, Denver, Colorado, November 29 - December 1, 1972*, pages 77–105. ACM, 1972.

[FFM+95] F. Ferrandina, G. Ferran, T. Meyer, J. Madec, and R. Zicari. Schema and Database Evolution in the O_2 Object Database System. In *Int. Conference on Very Large Data Bases*, 1995.

[FFM+01a] P. Fankhauser, M. Fernandez, A. Malhotra, M. a Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. http://www.w3c.org/TR/2001/query-semantics, June 2001.

[FFM+01b] P. Fankhauser, M. Fernandez, A. Malhotra, M. a Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. http://www.w3c.org/TR/2001/query-semantics, November 2001.

[FK99] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, Sept. 1999.

[FMZ94a] F. Ferrandina, T. Meyer, and R. Zicari. Correctness of Lazy Database Updates for an Object Database System. In *Proc. of the 6th Int'l Workshop on Persistent Object Systems*, 1994.

[FMZ94b] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *VLDB*, pages 261–272, 1994.

[FSS+97] C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, S. Zaniolo, C. Ceri, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.

[GB95] A. Gupta and J.A. Blakeley. Using Partial Information to Update Materialized Views. *Information Systems*, 20(8):641–662, 1995.

[GGMS97] D. Gluche, T. Grust, C. Mainberger, and M.H. Scholl. Incremental Updates for Materialized OQL Views. In *Proceedings of the Fifth DOOD Conference*, pages 52–66, December 1997.

[GM93] M.J.C Gordon and T.F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, 1993.

[GSW95] Y. Gurevich, N. Soparkar, and C. Wallace. Formalizing database recovery. In *COMAD*, 1995.

[GY98] J. Gross and J. Yellen. *Graph Theory and it Applications.* CRC Press, 1998.

[Har94] C. Harrison. An Adaptive Query Language for Object-Oriented Databases: Automatic Navigation Through Partially Specified Data Structures. Master's thesis, NorthEastern University, June 1994.

[HD91] M. Hardwick and B. R. Downie. On object-oriented databases, materialized views, and concurrent engineering. In *Proceedings of the 1991 ASME Int. Computers for Engineering Conference and Exposition.* Engineering Databases: An Engineering Resource, 1991.

[HMN+99] L.M. Haas, R.J. Miller, B. Niswonger, M.T. Roth, P. Schwarz, and E.L. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[HP00] H. Hosoya and B.C. Pierce. XDuce: A Typed XML Processing Language. In *WebDB(Informal Proceedings)*, pages 111–116. ACM Press, 2000.

[HS96] W.L. Hürsch and L.M. Seiter. Automating the Evolution of Object-Oriented Systems. In *International Symposium on Object Technologies for Advanced Software*, pages 2–21, Kanazawa, Japan, March 1996. Springer Verlag, Lecture Notes in Computer Science.

[IBM98] IBM Alphaworks. An Experimental Implementation of the Construction Rules sect ion of the XSL. http://www.alphaworks.ibm.com/tech/LotusXSL, 1998.

[Inc93] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.

[Jav96] JavaCC. http://www.sun.com/suntest/JavaCC/, 1996.

[Jin98] J. Jin. An Extensible Schema Evolution Framework for Object-Oriented Databases using OQL. Master's thesis, Worcester Polytechnic Institute, May 1998.

[Kau98] Rajesh Kaushal. View-based support for transparent schema evolution in federated database systems. In *International Workshop on. Issues and Applications of Database Technology*, 1998.

[KC88] W. Kim and H. Chou. Versions of Schema For OODBs. In *Proc. 14th VLDB*, pages 148–159, 1988.

[KC90] R. H. Katz and E. Chang. Managing change in a computer-aided design database. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 400–407. Morgan Kaufmann Pub., 1990.

[Kel82] A. Keller. Updates to Relational Database Through Views Involving Joins. In *Scheuermann*, 1982.

[KGBW90] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. *SIGMOD*, pages 393–402, 1992.

[KL95] G. Koch and K. Loney. *Oracle: The Complete Reference.* Oracle Press, Osborne-McGraw Hill, 1995.

[KR87] Bo Kähler and Oddvar Risnes. Extending logging for database snapshot refresh. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 389–398. Morgan Kaufmann, 1987.

[KR98] H. A. Kuno and E. A. Rundensteiner. Incremental Maintenance of Materialized Views in OODBs. *IEEE Transaction on Data and Knowledge Engineering,*, 1998.

[KR02] A. Koeller and E.A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views. In *Int. Conference on Extending Database Technology (EDBT)*, page to appear, 2002.

[Lau96] S.-E. Lautemann. An Introduction to Schema Versioning in OODBMS. In *Int. Conference and Workshop on Database and Expert Systems Applications*, pages 132–139, 1996.

[Lau97a] S.-E. Lautemann. A Propagation Mechanism for Populated Schema Versions. In *IEEE Int. Conf. on Data Engineering*, pages 67–78, 1997.

[Lau97b] S.-E. Lautemann. Schema Versions in Object-Oriented Database Systems. In *Int. Conference on Database Systems for Advanced Applications (DASFAA)*, pages 323–332, 1997.

[Ler96] B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report UM-CS-96-044, University of Massachusetts, Amherst, 1996.

[Ler00] B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.

[LLPS91] G. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to StarBurst: Objects, Types and Rules. *Communications of the ACM*, 34(10):95–109, 1991.

[LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *vldb*, 1996.

[LZLH94] L. Liu, R. Zicari, K. Lieberherr, and W. Hürsch. Polymorphic reuse mechanisms for object-oriented database specifications. In Ahmed K. Elmagarmid and Erich Neuhold, editors, *Proceedings of the 10th International Conference on Data Engineering*, pages 180–189, Houston, TX, February 1994. IEEE Computer Society Press.

[Mey92] B. Meyer. Applying "Design By Contract". *IEEE Computer*, 25(10):20–32, 1992.

[MIR93] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Int. Conference on Very Large Data Bases*, pages 120–133, 1993.

[MNJ94] M.A Morsi, S. Navathe, and Shilling J. On Behavioral Schema Evolution in Object-Oriented-Database System. In *Int. Conference on Extending Database Technology (EDBT)*, pages 173–186, 1994.

[MR83] L. Mark and N. Roussopoulos. Integration of Data, Schema and Meta-Schema in the Context of Self-Documenting Data Models. In Carl G. Davis, Sushil Jajodia, Peter A. Ng, and Raymond T. Yeh, editors, *Proceedings of the 3rd Int. Conf. on Entity-Relationship Approach (ER'83)*, pages 585–602. North Holland, 1983.

[MS93] S. Monk and I. Sommerville. Schema Evolution in OODBs Using Class Versioning. In *SIGMOD RECORD, VOL. 22, NO.3*, September 1993.

[MZ98] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 122–133. Morgan Kaufmann, 1998.

[NACP01] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. In *Proceedings of the ACM SIGMOD International Conference on Mana gement of Data, Santa Barbara, CA*, pages 437–448, May 2001.

[NLR98] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proceedings of International Conference on Extending Database Technology (EDBT'98)*, pages 359–373, Valencia, Spain, March 1998.

[O'B97] P. O'Brien. Making Java Objects Persistent. *Java Report*, 1(1):49–60, 1997.

[Obj93] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems.* Object Design Inc., December 1993.

[OPS+95] M.T. Oszu, R. J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. TIGUKAT: A Uniform Behavioral Objectbase Management System. *VLDB Journal*, 4:100–147, 1995.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *11th CADE, Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.

[PMD95] Joan Peckham, Bonnie MacKellar, and Michael Doherty. Data model for extensible support of explicit relationships in design databases. *VLDB Journal*, 4(2):157–191, 1995.

[PO95] R.J. Peters and M.T. Ozsu. Axiomatization of Dynamic Schema Evolution in Objectbases. In *IEEE Int. Conf. on Data Engineering*, pages 156–164, 1995.

[PR95] M.P. Papazoglou and N. Russell. A Semantic Meta-Modeling Approach to Schema Transformation. In *CIKM '95*, pages 113–121. ACM, 1995.

[QCR00] L.P. Quan, L. Chen, and E. A. Rundensteiner. Argos: Efficient Refresh in an XQL-Based Web Caching System. In *WebDB, Dallas, Taxas*, pages 23–28, May 2000.

[RAJB00] J. Roddick, L. Al-Jadir, and L.and et al. Bertossi. Evolution and Change in Data Management - Issues and Directions. *SIGMOD Record*, 29(1):21–26, March 2000.

[RCC00] E.A. Rundensteiner, K.T. Claypool, and L. et. al Chen. SERF-ing the Web: A Comprehensive Approach for Web Site Management. In *Demo Session Proceedings of SIGMOD'00*, 2000.

[RCL+99] E.A. Rundensteiner, K.T. Claypool, M. Li, L. Chen, X. Zhang, C. Natarajan, J. Jin, S. De Lima, and S. Weiner. SERF: ODMG-Based Generic Re-structuring Facility. In *Demo Session Proceedings of SIGMOD'99*, pages 568–570, 1999.

[RLN97a] E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proceedings of 4th*

Int. Workshop on Knowledge Representation Meets Databases (KRDB'97): Intelligent Access to Heterogeneous Information, pages 13.1–13.11, Athens, Greece, August 1997.

[RLN97b] E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proceedings of 4th Int. Workshop on Knowledge Representation Meets Databases (KRDB'97): Intelligent Access to Heterogeneous Information*, pages 13.1–13.11, Athens, Greece, August 1997.

[RLR98] E.A. Rundensteiner, A. Lee, and Y.-G. Ra. Capacity-Augmenting Schema Changes on Object-Oriented Databases: Towards Increased Interoperability. In *Object-Oriented Information Systems*, 1998.

[Roc00] Rochade Information Model. Rochade Information Model. http://support.viasoft.com/html/2RochDocDownload.html, 2000.

[RR87] A. Rosenthal and D. Reiner. Theoretically Sound Transformations for Practical Database Design. In Salvatore T. March, editor, *Entity-Relationship Approach, Proceedings of the Sixth International Conference on Entity-Relationship Approach, New York, USA, November 9-11, 1987*, pages 115–131, 1987.

[RR94] Y. G. Ra and E. A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Schema Evolution. Technical Report CSE-TR-211-94, University of Michigan, 1994.

[RR97] Y. G. Ra and E. A. Rundensteiner. A Transparent Schema Evolution System Based on Object-Oriented View Technology. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):600–624, September 1997.

[RS99a] A. Rosenthal and E. Sciore. First-class views: A key to user-centered computing. In *SIGMOD*, pages 29–36, May 1999.

[RS99b] A. Rosenthal and E. Sciore. First-Class Views: A Key to User-Centered Computing. *SIGMOD Record*, 28(3):29–36, May 1999.

[Run92] E. A. Rundensteiner. *MultiView*: Methodology for Supporting Multiple Views in Object-Oriented Databases". In *18th VLDB Conference*, pages 187–198, 1992.

[RW98] J.V.E. Ridgeway and J.C Wileden. Towards Class Evolution in Persistent Java. In *Third International Conference on Persistence and Java*, 1998.

[San95] C. S. Santos. Design and Implementation of Object-Oriented Views. In *Int. Conference and Workshop on Database and Expert Systems Applications*, pages 91–102, 1995.

[SCR00] H. Su, K. T. Claypool, and E. A. Rundensteiner. Extending the Object Query Language for Transparent Meta-Data Access. In *Ninth International Workshop on Foundations of Models and Languages for Data and Objects*, September 2000.

[SHL75] Nan C. Shu, Barron C. Housel, and Vincent Y. Lum. Convert: A high level translation definition language for data conversion (abstract). In W. Frank King, editor, *Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 14-16, 1975*, page 111. ACM, 1975.

[SHT+77] N.C. Shu, B.C. Housel, R.W. Taylor, S.P. Ghosh, and V.Y. Lum. EXPRESS: A Data EXtraction, Processing, amd REStructuring System. *TODS*, 2(2):134–174, 1977.

[SHT+99] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 302–314, 1999.

[Sjo93] D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.

[SKC+01] H. Su, D. Kramer, L. Chen, K.T. Claypool, and E.A. Rundensteiner. XEM: Managing the Evolution of XML Documents. In *ACM Research Issues in Data Engineering Workshop*, pages 103–110, 2001.

[SLT91] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-oriented Databases. In *Proceedings of the Second DOOD Conference*, pages 189–207, December 1991.

[STZ+99] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, and J.F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In Malcolm P. Atkinson and Maria E. Orlowska and Patrick Valduriez and Stanley B. Zdonik and Michael L. Brodie, editor, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[SWK+01] A. Schmidt, F. Waas, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and R. Busse. Xml benchmark project. http://www.xml-benchmark.org, March 2001.

[Sys01] Java Systems. The jaxp 1.1.1parser. http://java.sun.com, November 2001.

[SYU99] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents using Object-Relational Databases. In *Int. Conference and Workshop on Database and Expert Systems Applications*, pages 206–217. Springer-Verlag, 1999.

[SZ86] A. H. Skarra and S. B. Zdonik. The Management of Changing Types in an Object-Oriented Databases. In *Proc. 1st OOPSLA*, pages 483–494, 1986.

[TC98] P. Tarr and L. Clarke. Consistency management for complex applications. In *International Conference on Software Engineering*, pages 230–239, 1998.

[Tec92] Versant Object Technology. *Versant User Manual.* Versant Object Technology, 1992.

[Tec94] O_2 Technology. *O_2 Reference Manual, Version 4.5, Release November 1994.* O_2 Technology, Versailles, France, November 1994.

[TIHW01] T. Tatarinov, Z. G. Ives, A.L. Halevy, and D.S. Weld. Updating XML. In *SIGMOD*, 2001.

[Tr00] C. Trker. Schema Evolution in SQL-99 and Commercial (Object-)Relational DBMS. In *Ninth International Workshop on Foundations of Models and Languages for Data and Objects*, September 2000.

[UW97] J.D. Ullman and J.k Widom. *A First Course in Database Systems*. Prentice-Hall, Inc., 1997.

[VD91] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity and Inheritance. In *SIGMOD*, pages 158–167, 1991.

[Wut01] M. Wutka. The dtd parser. http://www.wutka.com, March 2001.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *SIGMOD*, pages 316–327, May 1995.

[ZLMR01] X. Zhang, W-C. Lee, G. Mitchell, and E. A. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *ICDE-RIDE 2001*, 2001.