MODELING CHANGES IN
INFORMATION SYSTEMS

by

Ranabir Gupta
B.E. (Hons.) Birla Institute of Tech. and Science, 1984
M. Tech. Indian Institute of Technology, 1986

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

© Ranabir Gupta 1995
SIMON FRASER UNIVERSITY
November 1995

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.
APPROVAL

Name: Ranabir Gupta
Degree: Doctor of Philosophy
Title of Thesis: Modeling Changes in Information Systems

Examinng Committee: Dr. J.J. Weinkam

Chair

Dr. Jiawei Han, Senior Supervisor

Dr. James P. Delgrande, Supervisor

Dr. Robert F. Hadley, Supervisor

Dr. Raymond T. Ng, External Examiner

Dr. Veronica Dahl, SFU Examiner

Date Approved: November 17, 1995
PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Modeling Changes in Information Systems.

Author:

Ranabir Gupta

November 15, 1995
Abstract

Databases are increasingly viewed as critical components in design and planning applications, where changes are integral to the domain. Traditional data models, however, enable the description of snapshots of a domain at particular points in time. Recent research has produced data models aimed at storing a series of snapshots, but reasoning with changes and with the interrelationships of objects as they change, typically requires means external to the database. This approach is not suitable for real-world applications where the relationships among objects in different dynamic states is of primary importance, as opposed to the static relationships at a particular point in time. Such applications demand data models to capture the integrity constraints associated with change histories.

This thesis investigates the use of techniques from formal languages to build models of dynamic information systems. Dynamic information systems model the structure and behavior of objects as their existence, observable properties, and states evolve over time. We propose a user interface language to model histories as first-class entities, and a computational formalism to reason with these models. Data structures are then proposed to aid reasoning with sets of histories. Extending dynamic information systems, we consider the case when the needs and world-view of an observer change over time, requiring changes to the representation of the domain (its schema) from her perspective. We thus propose the concept of adaptive information systems to reason with representations as an observers’ needs evolve over a period of time.
Acknowledgments

I am immensely grateful to my supervisor Dr. Jiawei Han, and members of my committee, Dr. Bob Hadley and Dr. Jim Delgrande, for their tolerance and support in allowing me to pursue a career while continuing to guide me in my research through constructive criticism, discussion and patient reading. I also thank Dr. Raymond Ng and the examiners for their comments.

This work, for whatever it is worth, would have been impossible without the support of my partner Gary Hall at Simon Fraser University, and Dr. G.M. (Frits) Swinkels at the National Research Council. I specially thank Dr. Nick Cercone, who shouldered many responsibilities during the oral candidacy and proposal, helping me in spite of his busy schedule. Bell Northern Research allowed me the use of their facilities to complete the draft, and I thank the management in particular for this kindness.

My gratitude goes out to other faculty members of the School of Computing Science for much valuable advice and the staff for much help and support throughout. The friends who supported me are too numerous to mention, but my gratitude matches your numbers. Finally, I beg the pardon of Parinaz and my family for the long wait.
# Table of Contents

**APPROVAL**  
iv  

**Abstract**  
iii  

**Acknowledgments**  
iv  

**Table of Contents**  
v  

**List of Figures**  
x  

## Chapter 1 Introduction  
1  

1.1 The Nature of Change  
2  

1.1.1 Our Working Assumptions  
3  

1.2 Philosophy Applied to Information Systems  
4  

1.2.1 Data Modeling Trends and Shortcomings  
4  

1.2.2 Thesis Outline and Contributions  
6  

## Chapter 2 Survey of Dynamics in Information Systems  
9  

2.1 Temporal Databases: Primitive Representations of Change  
9  

2.1.1 Relevance to dynamic and adaptive information systems  
10  

2.2 Data Structures for Temporal Information  
11  

2.2.1 The Grid File: A Common Basis for Temporal Indices  
12  

2.2.2 Seminal Data Structures for Attribute Oriented Timestamps  
12  

2.2.3 Tree and Descriptor based Temporal Indices  
13  

2.2.4 Searching an N-dimensional Space  
13  

2.3 Data Structures for Integrated Temporal Reasoning  
14  

2.4 Object Migration: Extending Temporal Databases  
15  

2.4.1 Object Migration: Overview  
16  

2.4.2 Other Approaches  
20  

2.4.3 Contemporary Grammar-Based Analytical Frameworks  
21  

2.4.4 Object-Oriented Updates  
22  

2.5 Relativizing Representations and Modifying Schema  
22  

2.5.1 Schema Evolution  
23  

2.5.2 Schema Relativization  
25  

2.5.3 Schema Integration  
27
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.4 Implementation Issues</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>2.5.5 Logics, Structures and Constraints</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>2.5.6 Peripheral Topics</td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>2.5.7 Recent Results</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>Chapter 3 Framework Architecture</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>3.1 Structure, Essential Modeling Devices and Features</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>3.2 High-Level Logical Architecture</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>3.3 Components of the Dynamic Modeling Framework</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>3.4 D'ALPHABET - Core Terminology for Dynamic Domains</td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>3.4.1 Attributes</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>3.4.2 D'Alphabet for Dynamic Descriptions</td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>3.5 Legality and Reactions</td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>3.6 Chapter Summary</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Chapter 4 D'DIALECT(O) - Object Dynamics Language</td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>4.1 Classifications Definition Language</td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>4.2 Essential Static Devices</td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>4.2.1 Bindings of Objects to Attributes and Classes</td>
<td></td>
<td>53</td>
</tr>
<tr>
<td>4.3 Attribute Semantics</td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>4.3.1 Dynamics-based classification of attribute functions</td>
<td></td>
<td>56</td>
</tr>
<tr>
<td>4.4 Reactions Definition Language</td>
<td></td>
<td>59</td>
</tr>
<tr>
<td>4.4.1 Preservation and Identity</td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>4.5 Essential Dynamic Devices</td>
<td></td>
<td>61</td>
</tr>
<tr>
<td>4.5.1 Transition</td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>4.5.2 Generation</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>4.6 Dynamic Features</td>
<td></td>
<td>68</td>
</tr>
<tr>
<td>4.6.1 Examples of Dynamic Features</td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>4.7 The Structure of Formal D'Language Definitions</td>
<td></td>
<td>75</td>
</tr>
<tr>
<td>4.8 Chapter Summary</td>
<td></td>
<td>78</td>
</tr>
</tbody>
</table>
6.8.4 Effecting Inference: The ATN Definition as a Recognizer ........................................ 164
6.8.5 Supporting Data Dictionary Functions ........................................................................ 172
6.9 Chapter Summary ........................................................................................................... 173

Chapter 7 Database Structures for Dynamic/Adaptive Info. Systems .................................... 174
7.1 Object and Historical Data Access Requirements ............................................................. 174
  7.1.1 Object-centered Access to Information ....................................................................... 174
  7.1.2 Class-centered Access to Information ....................................................................... 176
  7.1.3 History-Centered Access to Information ................................................................... 177
  7.1.4 Requirements due to Space and Schema Adaptation ................................................. 177
7.2 Introducing Dynamic / Adaptive Information Structures .................................................. 179
7.3 O-Structure: Framework for Dynamic Attribute Information .......................................... 181
  7.3.1 AVInt Tree: An Index on AV Intervals ....................................................................... 182
7.4 Summary Descriptors of Object States ............................................................................. 183
  7.4.1 Summary Descriptor Function .................................................................................. 183
7.5 C-Structure: A Framework for Dynamic Classes ............................................................... 186
  7.5.1 CM Intervals and Structures ..................................................................................... 187
  7.5.2 C-Structure .............................................................................................................. 189
7.6 Complexity of the Dynamic Information Access ............................................................... 195
  7.6.1 Choices in Interval Tree Structure ............................................................................ 199
7.7 Chapter Summary ............................................................................................................ 200

Chapter 8 Adaptive Information Systems: Extending Object Dynamics .................................. 201
8.1 Alternative and Changing Views on Data ......................................................................... 202
  8.1.1 Comparison of Object Dynamics and Schema Adaptation .................................... 204
  8.1.2 Framework for Adaptive Schema .............................................................................. 208
  8.1.3 Criteria for Schema Adaptation ................................................................................ 208
8.2 D'Dialect(S) Assumptions and Primitives ....................................................................... 209
8.3 Abstraction Mechanisms ................................................................................................ 211
  8.3.1 Restrictions in relation to Schema Adaptation ....................................................... 214
  8.3.2 Concealed Schema .................................................................................................... 219
8.4 Abstraction Dynamics and Meta-Reactions ................................................................... 220

-viii-
8.4.1 Dynamic aspects of Abstractions ................................. 221
8.5 Schema Change Reactions ........................................... 223
  8.5.1 Overall Schema-Structural Change ............................. 227
  8.5.2 Computational Aspects of Adaptive Schemas ................ 228
8.6 Schema Change Reaction Determinants ............................. 229
  8.6.1 Utility applied to Adaptive Schema ............................ 229
  8.6.2 Schema change using ATNs and Utility ....................... 232
8.7 Chapter Summary ...................................................... 234

Chapter  9 Conclusions and Future Work .......................... 235
  9.1 Future Work ......................................................... 236

Appendix A: Change and Perspective in Philosophy ............... 238
Appendix B: Utility ......................................................... 242
Appendix C: Augmented Transition Networks ........................ 245
Appendix D: Structures in Computational Geometry ............... 250
Appendix E: CSGs and Length-Reducing Histories in D'dialect(O) . 255

References ................................................................. 257
List of Figures

3.1 Static and Dynamic Relationships Expressed Graphically ................. 36
3.2 Dynamic and Adaptive Information Systems Architecture .................. 39
3.3 Structure of the Proposed Modeling Framework .............................. 41
4.1 Subclassification of Attribute Functions ..................................... 56
4.2 Essential Dynamic Definitions ................................................. 61
4.3 Essential Dynamic Definitions ............................................... 67
4.4 Spontaneous Reactions ....................................................... 69
4.5 Reversible and Irreversible Reactions ....................................... 72
4.6 M to N Reactions ............................................................ 73
4.7 Persistence Across Destructive Changes ...................................... 74
4.8 The "Null" or "Kill" Reaction .................................................. 74
5.1 Hybrid Modeling Guideline 1 ................................................ 81
5.2 Generalized Application of Hybrid Modeling Guideline 1 .................. 82
5.3 Hybrid Modeling Guideline 2 ................................................ 83
5.4 Hybrid Modeling Guideline 3 ................................................ 84
5.5 Beginning the Conceptual Design .............................................. 85
5.6 Adding Reapplications ......................................................... 86
5.7 Generalizing the Dynamic Behavior .......................................... 86
5.8 Completing the High-Level Conceptual Design ............................... 87
5.9 Film Processing Extended Data Flow Diagram ............................... 88
5.10 Film Processing ERD .......................................................... 89
5.11 D’LANGUAGE Film Processing Schema ....................................... 90
6.1 Register Arithmetic in ATN Traversals ........................................ 95
6.2 ATN interpretation of Classes .................................................. 100
6.3 ATN Interpretation of Transition Links ....................................... 101
6.4 ATN Equivalents of Generations .............................................. 102
6.5 Membership Specialization (During) ........................................... 104
6.6 Concurrent Branching in the Grammar ....................................... 105
6.7 Registers Categorized by Constraints Applicable ............................ 109
6.8 ATN Interpretation of Paper Production Plan ................................ 110
6.9 Interpretation of Adjective Semantics in a Grammar Framework ........ 121
6.10 ATN Interpretation of Spontaneous Evolutions ............................. 123
6.11 Spontaneous M:N Reactions in an ATN .................................... 125
6.12 Maintenance of reversibility in ATN ........................................ 128
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.13</td>
<td>Recorded and Derivation Histories</td>
<td>134</td>
</tr>
<tr>
<td>6.14</td>
<td>Example of a Linear History in a Query Domain Constraint</td>
<td>141</td>
</tr>
<tr>
<td>6.15</td>
<td>Example of Goal-Directed Operations</td>
<td>143</td>
</tr>
<tr>
<td>6.16</td>
<td>Non-determinism and Ambiguity</td>
<td>146</td>
</tr>
<tr>
<td>6.17</td>
<td>Implicit Use of Dynamic Definitions in Parsing mode for Queries</td>
<td>149</td>
</tr>
<tr>
<td>6.18</td>
<td>Complexity Estimates for Generator and Parser algorithms</td>
<td>161</td>
</tr>
<tr>
<td>7.1</td>
<td>Conceptual View of Evolving Classifications and Attribute Values</td>
<td>179</td>
</tr>
<tr>
<td>7.2</td>
<td>Practical Evolution of Attribute Values through Time</td>
<td>180</td>
</tr>
<tr>
<td>7.3</td>
<td>Restrictions on Attribute Value Evolution through Time</td>
<td>180</td>
</tr>
<tr>
<td>7.4</td>
<td>AV Int Tree</td>
<td>182</td>
</tr>
<tr>
<td>7.5</td>
<td>Evolution of a Summary Descriptor over time</td>
<td>185</td>
</tr>
<tr>
<td>7.6</td>
<td>CM-Ints in relation to AV-Ints</td>
<td>188</td>
</tr>
<tr>
<td>7.7</td>
<td>CM-Interval Example</td>
<td>190</td>
</tr>
<tr>
<td>7.8</td>
<td>AR-Tree and CM-Tree Example</td>
<td>190</td>
</tr>
<tr>
<td>7.9</td>
<td>Time-Independent Regions &amp; Class Containers</td>
<td>191</td>
</tr>
<tr>
<td>7.10</td>
<td>Physical Representation I</td>
<td>192</td>
</tr>
<tr>
<td>7.11</td>
<td>Object-Oriented Indices</td>
<td>193</td>
</tr>
<tr>
<td>7.12</td>
<td>Historical Indices</td>
<td>193</td>
</tr>
<tr>
<td>7.13</td>
<td>Comprehensive Hybrid Data Structures</td>
<td>194</td>
</tr>
<tr>
<td>8.1</td>
<td>Changes in Views over time</td>
<td>203</td>
</tr>
<tr>
<td>8.2</td>
<td>Views generated at one point in time</td>
<td>203</td>
</tr>
<tr>
<td>8.3</td>
<td>Comparing Object and Schema Dynamics</td>
<td>205</td>
</tr>
<tr>
<td>8.4</td>
<td>Abstraction Mechs and Meta-Reactions in terms of Restrictions</td>
<td>215</td>
</tr>
<tr>
<td>8.5</td>
<td>Attribute Role Types</td>
<td>222</td>
</tr>
<tr>
<td>8.6</td>
<td>Hypothetical schema change description</td>
<td>225</td>
</tr>
<tr>
<td>8.7</td>
<td>Example Central_OO_Schema from domain and user constraints</td>
<td>226</td>
</tr>
<tr>
<td>8.8</td>
<td>Data model evolution driven by incremental schema change</td>
<td>227</td>
</tr>
<tr>
<td>8.9</td>
<td>Schema Change through violation of a Use Factor</td>
<td>232</td>
</tr>
<tr>
<td>C.1</td>
<td>ATN Used In Natural Language Generation / Parsing</td>
<td>246</td>
</tr>
<tr>
<td>C.2</td>
<td>ATNs in Context-Sensitive Applications</td>
<td>248</td>
</tr>
<tr>
<td>C.3</td>
<td>ATNs in the Context Sensitive Expressive Space</td>
<td>249</td>
</tr>
<tr>
<td>D.1</td>
<td>Range Queries</td>
<td>251</td>
</tr>
<tr>
<td>D.2</td>
<td>Segment Trees</td>
<td>252</td>
</tr>
<tr>
<td>D.3</td>
<td>Interval Trees</td>
<td>252</td>
</tr>
</tbody>
</table>
Chapter 1 Introduction

The value of information often lies in its diagnostic and predictive value. Even a seemingly static description such as a photograph is difficult to isolate in the viewer's mind from the frames preceding and following it in time. This is particularly true if the picture is used as a diagnostic aid in investigating an event such as an accident, or as a planning tool for subsequent events. In many cases, a situation in time is useful only as an index into a series of events.

More so than photographs, information-bases representing situations at a point in time are used for diagnostic and predictive applications, which are intrinsically dynamic. Diagnosing the reason for a lack of spare parts in a just-in-time inventory system, or planning a preventive maintenance program in a telephone switch are two examples of applications that use large information bases to predict future scenarios or analyze past scenarios. Yet, until recently most information systems provided query and update interfaces only to static information; dynamic representation and reasoning was assumed to lie within applications using the data. However, dynamic modeling, querying and updating facilities are increasingly crucial components of the information bases themselves, as they are deployed in flexible design, control and creative applications.

This thesis investigates ways to specify and reason with dynamic domains and adaptive representations, in the context of a changing world and evolving user requirements. It proposes modeling primitives for dynamic domains and extends them briefly for adaptive schema. It is impractical in general for dynamic information to be represented exhaustively (as in a film) - the volume of information makes it difficult to find the significant information by scanning the analogue of a film in a database. We propose a way whereby a series of significant snapshots may be stored, and later used to extrapolate and intrapolate based on a definition of dynamic behavior. This yields advantages such as:

- Ability to capture plans and logical progressions as first class information entities.
- Ability to use snapshots as indices into the actualities, necessities and possibilities regarding the domain in the past and in the future.
- Ability to ensure the effective maintenance of certain dynamic constraints over changes in the domain and in its representation.
CHAPTER 1. INTRODUCTION

- Ability to tailor the representation of the domain to serve the information needs of its user over time, and thereby control the evolution of enterprise information storage.

Limitations must be imposed on this grand design. An infinite variety of changes may potentially occur, and the complexity of the dynamic constraints may cause them to be impossible to maintain effectively. One possible limitation is to identify and address only a small set of "common" change patterns. In the current stage of infancy of this field, we feel that assumptions of "common-ness" are unjustifiable. Instead, we propose a formal framework within which expressiveness of dynamic descriptions may be traded off against complexity and show that many practical dynamic domains may be described within the proposed framework.

We start by situating our framework in a philosophical context, emphasizing that the philosophical opinions reviewed are not strictly required to justify the limited sense in which change needs to be modeled in practice. We use the philosophical literature as an inspiration for our work, not for justification. We then review the dominant trends in data modeling and circumscribe the expectations from our data modeling framework, before introducing the framework itself.

1.1 The Nature of Change

From a philosophical perspective, potentially everything changes continually. It is the lack of change which must be explicitly axiomatized (as in the frame problem). Properties may change because of the passage of time (time-outs), as in the case of alertness changing into tiredness. We ignore these issues in this thesis on the following grounds:

- every fact in our framework is time-indexed, and those which can change must do so according to some explicitly stated rule with controllable complexity.
- consequently, the decidability of each time-indexed fact, or its converse through a closed-world assumption, is assured.

In other words, the frameworks we consider for modeling changes is relatively weak. Of greater significance to us are questions like "What constitutes a relevant and expected change to a user?" and "What triggers a re-classification of the domain information from a user's perspective?". The answers depend on the observers' notions of legal changes and change patterns on one hand, and observability on the other. Related topics like causality and empirical observation have received considerable attention in philosophy, as given in appendix A.
1.1.1 Our Working Assumptions

1.1.1.1 On Legality of Changes

In modeling change in practical domains, we are concerned more with legality of changes than with causality in the philosophical sense (see appendix A). The parts of the positive definition of causality which appeal to our intuitions regarding legal change include necessity, antecedence and repeatability, and also its rejection of deducibility from the definitions of antecedent or consequent states. Under these assumptions, legality becomes a prescriptive and experimental hypothesis to be tested, and a regime to influence and verify changes in the world. In information modeling, experimental validation of hypothesized legal dynamic relationships occurs through the users’ feedback regarding the categorization of objects resulting from changes in the world.

Once hypothesized, however, such legal relationships can be used to reason materially and modally about past and future states of the domain. Legal relationships between different object states define the accessibility relationships between possible worlds containing these object-states. Extending [164], the accessible worlds in the future are those which are obtainable through legal changes in the domain objects. These notions of legality are incorporated into the definitions of the modeling primitives proposed in this thesis.

1.1.1.2 On Observability and Representations

We start with the observation that not every representation of the world serves the needs of every application equally well at all times. The need to distinguish between two objects, and the categorization of discernible objects, depends on the observer’s needs and can change over time. Accordingly, the nouns and verbs used to describe the domain need to change, and the reasoning appears to be needed in the background with respect to the representation itself. Here again, we draw inspiration, without the strict need to justify, from the discussion in appendix A. As Quine illustrates: “Imagine a fragment of economic theory. Suppose its universe comprises persons, but its predicates are incapable of distinguishing between persons whose incomes are equal. The interpersonal relation of equality of income enjoys, within the theory, the substitutability property of the identity relation itself ...”. It is only in a background theory that has more to it than incomes that we can separate identities apart from their incomes.

Information systems largely model nominal kinds; even the most basic building blocks are not analogous to the physically or biologically founded natural kinds. Instead, they are
CHAPTER 1. INTRODUCTION

determined by the needs of the enterprise. Our framework for adapting schema to the needs of the observer is motivated by the coherentist philosophy for representations, specifically Rescher's pragmatic coherence, and by Quine's theory of ontological relativity. Coherence in this limited context is manifested in a reduced commitment to any foundational schema, and the ability to modify the schema, while maintaining coherence through compensating changes in other parts of the schema. In deciding which coherent schema is correct in a particular situation, we draw upon Rescher's theories regarding the pragmatic basis of truth relative to an observer's goals, while emphasizing that usefulness, not truth, is our concern.

1.2 Philosophy Applied to Information Systems

Information systems model very limited aspects of the world which are of practical relevance to information users. We interpret the theories as follows in the framework for dynamic and adaptive information systems proposed in this thesis:

- Legal relationships between states are not analytically decidable from definitions, but imposed. They may be used to verify changes, and to perform inferences in changing contexts. Deviations from the hypothesized legal relationships are usable in correcting the observer's view of the domain and its rules of conduct.

- Not all changes are significant, either legally or pragmatically. The main criterion for significance is pragmatic utility.

- A system of observations should be internally coherent in the sense that facts implied by a schema should not be contradictory as per the static and dynamic laws applicable.

- Different ontologies may be usable to describe each domain. Usefulness in achieving some practical objectives determines the appropriate schema for each observer.

- An observer's preferred ontology may change in step with her perceptions, knowledge, and the uses to which the information is put.

1.2.1 Data Modeling Trends and Shortcomings

Many newly emergent imperatives indicate a need for modeling formalisms interpreting concepts such as those above. For example:

- Control, planning and design applications are increasingly computer driven. Telecommunication networks, error-diagnosis systems in power plants, and
multi-agent production planning systems are examples. Common to these systems is a need to reason with objects and situations as they change over time.

- For dynamic creative processes like design and planning, large databanks are limited by today's inability to query it effectively. Temporal projection and inference, and customization are increasingly important.

- It is error-prone to represent the dynamic integrity of information apart from the information itself, i.e. within programs or minds. The need to capture it as knowledge is growing.

- Heterogenous information sources serving diverse communities of users according to their own needs need to interoperate. Information must be optimally represented and distributed across models.

The message is that alongside raw data, the meaning (or interpretability) of the information as it changes must be available to be interpreted appropriately. In recent years, data models have therefore incorporated more sophisticated semantics and associated operations, but the focus has been on the semantics of static worlds. Semantic and object-oriented data models (see [10] and [83], for instance) have yielded some useful features such as aggregation and inheritance. The object-oriented model even allows users to customize their data model through proprietary semantics in update and access methods.

Representation of changing information was initiated by temporal databases, which merely record change, omitting the intensional or legal relationships behind change, just as relational databases capture the incidental relationships among entities through foreign keys, omitting the semantics. Sophisticated users, such as production process planners, have other temporally-related concerns in dynamic modeling. A planner would manipulate a temporally-defined object such as the production plan for M rolls of paper from N batches of wood as one entity with attributes which are either temporally defined or based on non-temporal properties of plan elements constrained by rules by which entities progress through time such as average_machine_utilization and completion_dependencies for the last plan item. Current approaches to dynamic data modeling give inadequate support for such users:

- Many formalisms are process-centered (as opposed to class-centered): process-centered approaches are good for modeling attributes of individual functional steps in a sequence, such as average_wait_for_pulping. A user such as a planner views the world either in terms of the entities participating in a plan, which are complex objects, each of whose components has a temporal dimension, or complex plans (historical contexts), aggregated from many such complex temporal objects. Neither of these views corresponds to the process-centric approach.
Many schemes support atomic specification of dynamic constraints within the context of individual object types' lifecycles. This is inadequate in the context of plans, because attributes such as completion_dependencies are not associated with the lifecycle of a single entity, but are computable in relation to the states of other plan entities, recursively. Approaches which model individual object's lifecycles, with all external interactions modeled by opaque messages hide the dynamic semantics of the plan as a whole within the schema of a individual objects.

Often, participating objects can not be explicitly related in the context of the plan or historical context that they are a part of. If so, utilization might be computed by the planner from an undifferentiated time_of_arrival user-defined attribute attached to waiting_pulp_batch entity-type (among others). Neither attribute nor entity correspond to planner-identified objects or to process steps (see chapters 2 and 5). Changes to a plan or historical context must also be explicitly computed when a participating object changes.

Modeling tools should extend to a variety of relationships and indicate the difficulty of reasoning with the resulting models. For instance, legal dynamic relationships often relate M prior states to N consequent ones, imply reversibility or other domain-specific semantics. Existing schemes are often limited by their ad-hoc nature of the primitives to particular problem domains, and their computational implications are not evident.

In terms of modeling, objects may concurrently play more than one role and participate in more than one dynamic relationship. They may be created and destroyed. Many existing schemes do not address mechanisms for such behaviors.

Finally, we recognize that static and dynamic schemas need to change over time depending upon a user's changing needs. In fact, even the change in an object's state is indirectly related to the fact that it's existence in the previous state is not useful to the observer. This factor is not directly addressed in existing research.

As a result of these shortcomings, static and dynamic aspects of a domain are typically modeled in isolation. This thesis proposes a framework which addresses the shortcomings outlined above.

1.2.2 Thesis Outline and Contributions

Our overall objective is to present a top to bottom picture of how to model and reason with dynamic domains and schema in information systems. After reviewing some of the existing research in this field in chapter 2, we propose a minimal framework that allows the separation of some crucial concerns in dynamic modeling in chapter 3. In the context of our overall goal, the main contributions of this thesis are the primitives for modeling dynamic
behavior of objects as introduced in chapter 4, and their interpretation in terms of augmented transition networks (ATNs) in chapter 6, including query and update semantics. In support of this interpretation, we adapt existing ATN algorithms and hybridize existing data structures from computational geometry in chapters 6 and 7. Our adaptation, and the application of ATNs to modeling, querying and update of dynamic information are also significant as contributions, even though we do not claim optimality in this area. Beyond these main contributions, we address some modeling principles in chapter 5 to show the potential of our primitives as a modeling discipline, and also illustrate the extension of our concepts to dynamic schema in chapter 8. Both of these contributions are incomplete, though we regard the extension of concepts in chapter 8 to be significant as well. Chapter 9 presents the conclusions and topics for future research.

Both the ability to model dynamically-defined objects such as plans, as well as the need to meet specific performance criteria, pose important problems. In this thesis, we focus on the first problem, investigating modeling mechanisms. Towards the second problem, we indicate the bounds on performance implied by our proposed modeling mechanisms, and argue for their adequacy, but not their optimality. In fact, we point out that these limits can be improved, particularly if the modeling problem is restricted. In proposing modeling mechanisms, we avoid entirely new theoretical abstractions, applying existing formalisms wherever possible, so that the body of existing results and tools may be adapted for use. Data modelling is the art of interpreting the real world, which is a subjective task. If the set of abstractions provided is either too weak, or large and rigid, users would need to circumvent them and represent semantics outside the framework, obviating its purpose. Therefore, the framework proposed here provides:

- A small set of domain-independent dynamic modeling abstractions: This allows us to model entities such as plans (historical contexts), which exist as first-class objects with temporal and non-temporal properties. The relationships among states of individual objects in the context of the plan they participate in can be made explicit, along with the relationships among their attributes. Features such as multiple concurrent class memberships, and M:N and reversible relationships can be modeled based on our intensional interpretation of membership in groups.

- A computational framework in terms of which the extensions and restrictions mentioned above are specified, implemented and analyzed: The framework encapsulates the semantics to reason transparently with temporal semantics, much as object-oriented inheritance semantics reason transparently with certain
static relationships. Since it is based on formal languages, it has well understood complexity and expressiveness with respect to sequences of states over time.

- Data structures and algorithms which enable set-based operations for information systems: These help to establish the complexity implied by our framework. They extend equally to objects and schema, and incorporate the notion of utility so that changes may be effected in response to changing needs.

Thus, our proposals substantially address some major shortcomings in the existing research on modeling dynamics. We first address dynamic relationships in a domain with a fixed ontology under the name of dynamic information systems. We extend these concepts briefly to illustrate the framework’s usability in reasoning with schema change in response to users’ needs. The latter is referred to as an adaptive information system.
Chapter 2  Survey of Dynamics in Information Systems

Data models have evolved over time from flat file systems through ER model [39] to semantic and object-oriented data models [75][83]. Brodie [31] offers a good discussion of the motivations and early developments in this area. The object-oriented model (see e.g. [10], [151], and [136] for concepts of object-oriented systems design) combines the richness of semantic models with the concept of a single modeling primitive - the object. Recent efforts have been directed at finding a good semantic basis for object-oriented databases, most of them based on logic [15][94]. There have been several proposals regarding semantically well-founded database languages integrating logical and object-oriented features [2][3][174], including efforts at analyzing the computational complexity of operations based on such formal models. Object-oriented databases provide a powerful computational framework, limited only by building restrictive class structures (schemas). Thus we have research fields such as 'geographic' and 'geometric' data models. Our focus in this thesis is on defining similar limiting object-oriented schemas for dynamic and adaptive information systems. We survey some results below.

2.1 Temporal Databases: Primitive Representations of Change

Changing situations were first archived in temporal databases, now a mature field with:

- Terminology regarding the kinds of time representable in databases.
- Definition of algebras operating over time periods and intervals.
- Initial data structures required for efficient reasoning with time points and intervals.

Temporal databases are indexed archives of all states of the world over time. ([27] is a comprehensive review up to 1982.) Temporal database primitives are primarily of three kinds [153]. Queries such as "When did Mary become a teacher?" involve user-defined time, which is indistinguishable from other attributes, and has a readable format like the Gregorian calendar.

More significant from the data modeling perspective is the need to reason with temporal relationships, as in the query "Who were the Ph.D. students during Mary's professorship?". This question involves reasoning with valid time, the actual time when asserted facts held in the
universe. Valid time may not be user-readable, since it exists to enable efficient reasoning with time; it could be represented, for example, using "seconds since 1/1/1971" and event chains.

A third kind of time investigated is transaction time, the time when the database became aware of events. Transaction time has been investigated with respect to its interaction with valid time, to relativize queries to the knowledge contained in the database at particular times. Queries such as "When the British pound fell, what was known about the selling of British Government bonds by the Deutschbank?" are based on the presumption that the transaction time of a Deutschbank sell-off could be later than the valid time of the British pound's fall. Such considerations are important in the context of long (design or planning) transactions. However, we concern ourselves solely with valid time in this thesis.

Researchers have also investigated temporal relationships (e.g. [38]) and algebras based on the above notions of time. Common ontologies are founded on timepoint- or interval-based state-spaces, structured as datelines and before-after chains. The expressiveness and complexity of reasoning with temporal operators and predicates such as during, initiates and comparable has been investigated. Logic-based models such as the situation calculus [5] have been used as semantic foundations, and object-oriented [7] or extended relational implementation schemes [63][152] have been proposed. Several temporal query language extensions of relational languages, e.g. TQUEL, have emerged as standards for implementation.

Recent research in temporal databases typically concerns the extension of these primitives to approximate and fuzzy operators. The semantics of time and change in databases has been based primarily on temporal logics [99], and a good account of temporal concepts such as 'before' and 'forever' based on intensional logic has been given in [48].

2.1.1 Relevance to dynamic and adaptive information systems

To reason with legal temporal relationships, time points or intervals as well as predicates such as overlap and intersect must be admitted into the vocabulary. We thus adopt the half-open interval representation suggested in [5]. Our query language also resembles those proposed in temporal databases. Beyond these similarities, temporal databases do not start to address the concerns of dynamic and adaptive information systems. For example:

- Temporal database schemas are knowledge-poor - semantic relationships among attributes of the same or different entities are absent.
The changes represented in temporal databases are incidental; Any legal relationships which they may have with other changes must be external to the semantics captured by the temporal database itself.

Temporal inference, active update or monitoring are not supported by temporal databases.

Due to the lack of schema semantics, and lack of representability of expected change patterns, integration of static and dynamic semantics during data modeling is impossible; so is modifying a schema in response to changes in the users' needs.

To summarize, we borrow some definitions of intervals, temporal relationships, and query primitives from temporal databases, but add the modeling of static/dynamic semantics.

2.2 Data Structures for Temporal Information

The development of data structures and algorithms for efficient temporal reasoning lags the research in temporal algebras. Recently there has been increasing activity in this area, since relational techniques do not serve the needs of special purpose temporal reasoning well.

Ling and Bell [105] give a review and bibliography of the design and implementation of time models. Different implementations associate the temporal dimension with relations, tuples, or individual attribute values (in relational systems). The former schemes are simpler, but often lead to the archival of redundant information because timestamps are associated with unchanged information. Attribute-based time-stamps are efficient in space, but require more sophisticated data structures, which some recent investigators have concentrated upon.

In interval-based implementations, indices are usually geared towards search based on inclusion, while in point-based models, searches are indexed stabbing queries. It is interesting that hybrid schemes are generally absent.

The area of multidimensional range search in temporal databases has not been well-covered. Briefly, some shortcomings are that changes in class definitions are not anticipated, insertions and deletions of intervals are not well-studied, complexity is sometimes not analyzable, and hybrid range and interval based schemes are rare. In the area of computational geometry, problems involving segment queries in conjunction with range queries have been addressed. Some results which influence our own proposal are summarized in appendix D.
2.2.1 The Grid File: A Common Basis for Temporal Indices

The grid-file [122] is a common temporal index. Grid files assume domains with:

- Few attribute dimensions, with a large and dense range of values in each dimension.
- All attribute dimensions are symmetric, or used identically.

Grid files partition the embedding space into n-dimensional grids. Each grid is mapped N:1 into buckets, which store fixed amounts of information. If grids overflow, then new grids are created, but not new buckets. If buckets overflow, then new grids and buckets are created. Buckets (and grids) divide and recombine using buddy or neighbor schemes. If these assumptions are violated, or if the ratio of grids to buckets is small, the grid file is inefficient.

Variations of the grid file include schemes to make some attribute dimensions "merge only" (or "split only") so that the granularity of (asymmetric) grids along different dimensions decreases (or increases) monotonically. We adapt this dynamic aspect of grid file re-organization to support the dynamic creation and modification of secondary indices for regions of the attribute space associated with classes in an adaptive information system.

2.2.2 Seminal Data Structures for Attribute Oriented Timestamps

Shoshani and Kawagoe [149] propose the seminal data structure based on attribute-oriented timestamps, using a 2-dimensional grid-file of start-time and duration. A region search on this grid is used for temporal queries. This scheme does not address domains with irregular, step-wise-constant, dynamic temporal intervals (as in dynamic and adaptive information systems) well. What we do adopt in our proposal is attribute-based time stamping, which conceptually associates a single time interval with each <attribute-value, object> pair. The use of grid-files as the primary indexing mechanism, and the use of atomic, non-overlapping time periods for each object state in [149] causes more than one physical time interval to be associated with each logical <attribute-value, object> pair, making storage inefficient.

Rotem and Segev [135] improve the efficiency of schemes such as the above through asymmetric grid-files, which distribute tuples among the grid blocks better. However, their structure is asymmetric in all dimensions except temporal, because of their assumptions about data accesses. We propose data structures for our domain with the converse assumption.
2.2.3 Tree and Descriptor based Temporal Indices

In the area of conceptual structures, [106] introduces a tree-based temporal index with separate trees for 'past' and 'current' facts and secondary links among the leaves of the tree. A variation of tree-based temporal indices combining all tenses in a single tree, in combination with the grid-file schemes discussed in the previous section comprises our proposal. An alternative way to index multidimensional asymmetric spaces involves extensible hashing and descriptors. [59] initiated discussion on extendible indices, which trade off the number of levels of indexing with the amount of time taken to search an index (O(1) if the hash value leads directly to a bucket, O(n) if too many buckets have to be chained, O(log(n)) for index trees). The authors use a variation of tries (radix search trees) where a single level of the trie (the hash value) could encode a variable number of attributes. If the height of the tree is acceptable, and the number of entities per bucket match the bucket capacities, then the radix tree could encode the hashing of one attribute at each level. If, on the other hand, the number of entities per bucket is below bucket capacity, and the height of the radix tree is excessive, then each level of the tree could be modified to encode more than one attribute. We utilize the idea of extendible descriptors as a 'summary' field which narrows the non-temporal search in the context of changing class descriptions.

2.2.4 Searching an N-dimensional Space

The non-temporal attributes of a domain form a multi-dimensional space within which n-gons represent classes and points represent objects. Queries regarding class memberships can then be modeled as point-containment queries in the n-dimensional space. Range searching techniques in multi-dimensional spaces are therefore usable for the non-temporal aspects of a domain. R-trees, which extend B-trees for multi-dimensional spaces, divide the search space into (possibly) overlapping boxes. Our proposed data structure for non-temporal aspects of queries resembles the R+ tree [145], which uses a distinct region to represent the intersection of two or more rectangles, instead of allowing overlaps. Space utilization is 60% in R+ trees (10% lower than in R-trees) but since regions do not overlap, searching is cheaper. Other indexing schemes which are customized to object-oriented data models with regard to their generalization and aggregation semantics are given in [22] and [108]. We consider these schemes to be peripheral to the topic of modeling dynamics in information systems.
2.3 Data Structures for Integrated Temporal Reasoning

The Molecule Atom Model (MAD) [90] is the most comprehensive scheme combining temporal and non-temporal attributes and reasoning into a "temporal complex-object data model". Designed for domains where a single entity may be defined over many tuples over time, MAD relates entities using an ontology of atoms and molecules linked by pairs of REF-TO attributes e.g. Employee.Clubs and Clubs.Members. With each atom state, they associate a valid from attribute, a valid to attribute, a past attribute, and a future attribute. The two latter attributes are used to link the histories of each atom. Queries are defined to be juncture (at a point in time), existential (during an interval) or universal (throughout an interval) queries. It is recognized that in updates, 1:1 relationships between atoms and molecules become N:1 relationships between "states of atoms" and "states of molecules". Though well conceived for storage of changes, MAD retains some deficiencies common to previous models, and more:

- Update-only databases are allowed; deletion is a difficult operation.
- In common with other schemes, devices for representing and reasoning with change are absent.
- The data structures do not support efficient query processing or update. Updating a value at a particular time needs a sequential search of the validity intervals for the attribute.
- The semantics of the scheme are restricted by the MAD model. For example, all links between atoms and molecules carry identical, predefined semantics.
- Duplication of information in atoms between atom-states leads to an artificial enforcement of referential integrity.
- Time lines are strictly linear, making the system optimized for history keeping, but not for maintaining the status of versions, which require concurrent branching time lines.
- Atom types are fixed, and no schema evolution or relativization is possible.

Qu et al in [130] define a system where objects belong to different sets of predefined classes during successive periods of time. The classes to which an object may belong are organized in a generalization tree. A timespan denoting an object's time of membership in the corresponding class is associated with each node of this tree. Such history records are linked to previous and following history records. The special contribution of this scheme lies in:
The clustering techniques defined for storing records of membership in time of the memberships in the classes. Thus, for example, a cluster a1a2b1b2b3c1 may denote an object's membership in classes a1 and a2 successively over time t; if these are superclasses of b1, b2, b3, then the following record denote successive memberships over the same period t.

The partitioning of the class composition tree, and the clustering of time intervals with respect to reference intervals. These induce a clustering on the data, which is indexed by time and class.

This scheme represents significant progress, but in common with other proposed schemes, the authors do not consider issues of reasoning with expected change patterns, schema evolution or versioning. The practical implications of O(nlogn) storage are also high, since the number of intervals is two orders of magnitude larger than the number of objects.

Dayal and Wuu [52] also propose a storage scheme for temporal data at the level of class memberships in predefined classes. The change points where an object changes its membership in a class are organized in a B+ tree. The leaves of the tree store each interval which overlaps the change point represented by the leaf. This implies a high storage cost and insertion costs, because most intervals cover more than one change point.

2.4 Object Migration: Extending Temporal Databases

Temporal databases offer means to record the states of a domain over a period of time; the integrity constraints to be maintained over changes, and reasoning with these changes, are not of significance. Temporal databases may be enhanced by systems which model processes of change in addition to storing and accessing the change history. Such systems would motivate users to model both the static and dynamic aspects of a domain within one system, obviating the need for private applications which hide parts of the semantics. There are at least two kinds of change which could be modeled and managed by an information storage system:

- Change in the facts which hold in a domain of discourse over time. These changes may manifest themselves through update of an object's attributes, or in the creation, deletion and migration between categories (classes) of objects (Chapter 3).
- Change in the representation of a domain within a database system. This would reflect changes in the use of the information, as well as changes in subjective preferences of the user, and objective constraints about the world which need to be preserved (Chapter 8).
2.4.1 Object Migration: Overview

The emerging field of object migration addresses modeling tools and formalisms for the maintenance of integrity constraints in a dynamic world which necessitates frequent updates to the databases which model the world. The questions addressed include:

- What are the dynamic relationships among objects? (histories, versions ... etc.)
- What modeling primitives are needed for such relationships and what are their properties?
- (Subsequently) Is there a uniform dynamic framework for objects and schema?

Under the topic of object migration, we find diverse concerns. In general:

- The research is largely experimental. Many proposals are not formally defined or analyzable, and can be compared only in subjective terms in the absence of canonical representations.
- Research aims range from improving static descriptions through knowledge of the dynamic behavior to design of transactions. In general they do not attempt to differentiate aspects which are basic to dynamics from add-on semantics which vary from one application to the next, making them non-extendable beyond the pilot applications.
- At one extreme, process-oriented frameworks regard change as being of primary interest, according the objects involved a second-class status. At the other extreme, proponents of object-oriented change localize the specification of change within each object definition. Thus, changes involving multiple object-states are hidden.

We propose a scheme to extend object-oriented methods with the concept of a formally defined history object, encapsulating the definition of change over time. Such a representation is useful for design or planning domains [171]. We classify the previously proposed schemes into three broad categories as below.

2.4.1.1 Transaction and Process Modeling

The first body of work is aimed at deriving separate static and dynamic descriptions of a domain in a coordinated manner (e.g. [34], [121], [107]). Two distinct subgroups exist: transaction modeling and process modeling. Transaction modeling is concerned with the design of database transactions in harmony with the physical data representation. The Active and Passive Component Modelling (ACM/PCM) methodology [34] for transaction modeling starts with a static description. Depending on the descriptions of the changes to the static picture, it proposes a method to design consistent transactions on the static structure.
Ngu [121] rejects the restrictions imposed by a fixed static structure, and emphasizes the strategy of conducting transaction modeling prior to the completion of the design of the static schema. Starting with an incomplete static representation of the world, Ngu gives a method to refine that picture incrementally by anticipating and considering the expected ways in which updates may be done. Thus, objects and static relationships referred to or implied by specific transactions are added to the schema, leading to a second phase of static schema design. The result of this methodology is a specification of the preconditions and actions involved in a transaction, in conjunction with a good static description of the domain.

Some earlier work ([138], [57]), attempts to integrate static and dynamic views of a domain using Entity-Relationship (ER) diagrams [39] and Petri Nets [126] respectively. The two views are correlated through objectified time periods. Some of the entities created by this method are unintuitive (e.g. "Student-of-Semester" and "Study-of-Semester-of-Study-State", p. 122), and nevertheless yields two different representations. Transaction modeling, in summary, is a tool for optimal design of the low level representations of data, to facilitate the intended data manipulation. Our insights from transaction modeling include the usability of preconditions and associated actions in defining the semantics of change.

Process modeling systems consider processes to be the primary carriers of information. They start the construction of the conceptual schema from some Data-Flow diagram. For example, the event model [96] structures process events hierarchically into super- and sub-processes by function and communication links which transmit information from event to event. The dynamic schema is transformed into a schema containing application events which model database transactions. In Markowitz’s [107] proposal, process nodes in Data Flow diagrams are converted into relationship sets or entity-sets in an extended ER conceptual schema. Scripting-based systems (e.g. [32]) use adaptations of Petri nets to enforce dynamic integrity constraints and define user interactions. Since process modeling systems are primarily aimed at deriving efficient static schemas, they do not consider integrated data models usable in preserving dynamic integrity constraints and inference of states over time.

A Critical Evaluation: Transaction modeling constructs are meant for low-level design, not for system-level conceptualization, integrity maintenance, or inferences about past, present, or future states. In effect, if a transaction such as "insert a record of John as a M.Sc. student" arrives at the system out of the blue, transaction modeling is adequate to specify the details of that insertion in terms of the creation of records of John as student, person etc. In most appli-
cations, more information about the progress of an entity through time is available. E.g. John must have been, but is no longer, an applicant either for the M.Sc. or Ph.D. programs. If he completes the program successfully he will become an alumnus, otherwise he will become a "Non-Completer". If he becomes an alumnus, he will always remain one but may apply to the PhD program. These all (possible) evolution paths of John. Further, transaction processing is of no help in inferring, for example, the english proficiency of John from his TOEFL score in the absence of other measures, such as his average in English courses. This activism in inferring states and performing checks is impossible with the limited information in transaction model.

Process modeling is useful for high level modeling of data-flow dynamics. Thus it is successfully employed for modeling data processing systems. Its drawbacks are the same that motivate object-oriented systems - change in a process necessitates widespread change in many elements of an implemented system representing the process. Moreover, in the case of evolutionary object-oriented changes, information is not passed from process to process; rather entities change from class to class. It is also unsuitable for the inference and activism mentioned above, which need the integration of the dynamic and static aspects of the model.

2.4.1.2 Petri Nets

The second distinct body of work enhances Petri nets (see [126]) with structured entities ([128], [91], [88], [80] for building analyzable, active systems for automatic update consistency maintenance and inference. Briefly, Petri Nets have 'places' connected M:N by 'transitions'. Tokens occupy places, and when all the input tokens for a transition are present, transitions are spontaneously fired, causing tokens to be placed in the output places. Petri Nets represent data-driven, parallel, non-deterministic systems which address reachability concerns well. Tokens and places lack internal structure, making Petri Nets primarily a parallel process-centered formalism. In object-centered domains, where the relationships within and among individuals are semantically rich, and incremental changes to the internal structure of individual objects (sometimes) cause macroscopic changes in their representation and classification, Petri Nets prove semantically poor. For example, the promotion of a student from one grade to another either upon reaching a credit limit or upon getting some CGPA, is not specifiable.

The Petri Net formalism needs semantic enhancement for representing such changes. Recent attempts at such a modification [88] (Colored Petri Nets) to add structure to tokens and
places do not integrate static and dynamic models well, since no notion of patterns of transitions is native to Petri Nets. In the opinion of its creators, CPNs lack analyzability beyond a point for modeling incremental change and reasoning with historical data objects. In summary, Petri Nets are ideal for system studies involving attribute-less, asynchronous and parallel changes, but not when complex object structures and histories, along with semantically rich relationships are involved.

2.4.1.3 Distributed Migration Descriptions

The third body of work comprises systems to model the migration of objects in an OODB among classes or roles ([134], [165], [125], [58]). In intent, this body of work is closest to our own interests, because it assumes that in the course of its lifetime, an object may go through a sequence of states of existence. A description of these changes, and the conditions which trigger them, are encoded with the class definition and used to flag the object's existence in particular states. Some object-oriented data models assign state transition automata directly to classes e.g. [134], or roles [125] which in turn belong to classes. Accordingly, an object exists in various roles or states within its class. A serious shortcoming of these systems, arising from an object-oriented virtue, is the fact that they do not allow encapsulation of information about distributed changes (i.e. they localize the description of allowable changes to a single instance of a class). If a useful migration pattern involves two or more distinct objects from the same or different classes, the description of that change is not encapsulated at one place, but distributed in the definitions of the many classes involved. This has two major consequences:

- Either it disallows objects from migrating between classes (an object can not be a member of one class A at time t, and later start being a member of a different class). If such migrations are to be allowed, all entities must be members of a single large class. Both options are clearly unacceptable, since ordinarily people recognize applicants and students to be distinct classes, to be represented as such, and also recognize that applicants sometimes become students.

  In an OO schema, it may be argued that since both applicants and students are instances of the class person, and the change should be associated with the definition of persons. This is the OO equivalent of the relational universal relation assumption, which has fallacies.

- The assumption that change in state or role of an object depends solely on factors local to the object places a fundamental limitation on reasoning with past or future changes. Many real changes involve more than one object. For example, a batch of sulfuric acid comes into existence as a result of the concurrent existence of batches of water and
oleum. [125] does provide means to correlate instances of different classes, but the fundamental restriction of state changes being specified by locally situated finite automata makes the mechanism cumbersome.

We believe that the extension needed to remedy these shortcomings consists of a means to refer to the states of objects in different classes (or multiple numbers of objects in the same class) while specifying the applicable migrations.

### 2.4.2 Other Approaches

[91] takes the middle ground between object-life-cycle specifications and Petri Nets. It modifies PNs to make transitions user-mediated, rather than spontaneous. In our opinion, this is an effort to adapt a formalism to tasks which it is not best suited for. [58] approaches the question of object migration from the perspective of updates to object-oriented databases. Object-oriented updates could cause the updated object to become a member of a class specified by a more general or more specific type definition. From this motivation, the author develops the possibility that certain updates might cause objects to relocate to classes which are not direct generalizations or specializations of the original class. Since the prime motivation concerns object oriented updates, however, the algorithms are tied to the idea of a graph traversal in a fixed generalization hierarchy. While similar to the system of El-Sharkawi and Kambayashi, we propose an analyzable system which is more powerful in allowing an object to belong to more than one class at a time, allowing generation and extermination of objects, M:N reactions and other features useful in practice, along with a facility to reason with change.

More recently, Falkenberg, Oei and Proper [60][61] propose a set of primitives such as recording, correcting and forgetting, to model the change of entities and their migration among classes over a period of time. Though comprehensive, their system suffers from its ad hoc nature and consequent lack of analyzability. It is difficult to model M:N changes, the act of forgetting is unanalyzable because of its isolation from other changes, and the mutual exclusion and exhaustion of expressive coverage of their primitives is difficult to determine.

Tan and Katayama [156] and some other authors (e.g. [102]) have specified systems which are specific to domains such as software modules' evolution and medical cases' progress. Though their ideas about the structure of software modules can be extended to general evolving objects, the dynamic primitives defined are domain-specific in themselves.
2.4.3 Contemporary Grammar-Based Analytical Frameworks

Attempts at developing frameworks for analysis of the systems proposed have recently emerged. Our own work ([74],[67] and later work), along with Su [155], represents the first attempts at recognizing the expressive power and computational complexity of systems for object migration. Su's work embodies the same intuitively attractive idea which had motivated our own research, namely that a sequence of classes to which an object belongs over time is akin to a sentence over a set of symbols in space. Allowable sequences are thought of as a language; coverage of that language and computability/complexity are analyzed in terms of the grammar for the language.

Su calls the set of classes to which an object may belong at one time a role set. A migration pattern is a sequence of role sets. A migration inventory is a set of migration patterns which is closed under prefixes. A particular migration inventory may represent the allowable sequences of role sets (a dynamic constraint) which an object may undergo in its history. A migration pattern is viewed as a word over the alphabet of possible role sets. A migration inventory, then, is a language of such words.

Su provides a basic transaction language, SL, with operators which act upon subsets of the instances of a class which obey some select condition. It is proved that SL generates all and only regular migration patterns. This follows because each object behaves independently of every other object. Moreover, a migration depends only on an object's current class membership.

SL is extended to CSL(+) in [155] by the addition of conditional updates, defined as $d_1, d_2, \ldots d_n \rightarrow t$, where the $d_i$'s are literals and $t$ is an atomic transaction. CSL(+) transaction schemas are strictly recursively enumerable because of the conditions are unrestrained and can connect any two classes whatsoever. Su's results imply that in general, it is undecidable to say if an object in class $P$ obeying a condition $c$ can migrate to class $Q$ obeying a condition $d$.

Su's framework is theoretically oriented (not motivated by practical modeling needs) and does not consider the costs of expressing and implementing specific practically useful features. This thesis defines a framework for practical systems which occupy the middle ground between Su's extremes.


2.4.4 Object-Oriented Updates

Object oriented update frameworks are the subject of many recent research efforts. In order to express change logically, many authors represent objects using an F-Logic like language [94]. In order to avoid updating the axioms of the domain, every update logically creates a new version of the updated object, strictly, of a different type; this concern with preserving the logical semantics thus leads to pragmatically dubious choices. For example, the preservation of identity in this framework (is a new version of an object the same object, or a different one?) is difficult. In a strongly typed environment, if a new version is viewed as new type creation, the methods which were defined for the existing object need to be redefined in order to be defined on the newly created version of the object after every update. Though we use logic to reason with some constraints in our proposal, we rely on a grammatical framework and model to express the dynamic behavior, rejecting a logical definition of the entire dynamic system. Dong and Li [56] also propose primitives for object migration without attempting completeness with respect to any model. Their main concern is the set of conditions under which the migrations are information preserving and the reachability of some classes from others through migration paths, which are not the main focus of our research.

Object migration implementation is epitomized in T AXIS [114]. T AXIS defines a meta-model to express database schemas and transaction schemas. Transactions can specify pre-conditions and actions associated with their execution. Apart from its comprehensive implementation, however, T AXIS suffers from the same limitations as a number of similar systems. It lacks formalization for inference and analyzability. Further, the meta-classes definable in the framework are useful for schema definition, but lack from the point of view of reasoning with schemas, besides being an ad-hoc set not justified with respect to sufficiency.

2.5 Relativizing Representations and Modifying Schema

In data modeling, distinctions are drawn between objects, classes and the roles they play (see [170]). These are justified as follows: person is a class because if an instance exists, it exists as a person. Student is a role because if an object is a student, it may continue to exist even when it is not a student. A role is payed by an object in a certain state: students and employees are persons in certain states. However, these distinction become weak when one considers that some roles can be played n times simultaneously (a person can be two employees) and roles
can be played by other roles. If a house is a class, how do we classify a house which is used as a shop later and an office after that? Perhaps house is a role of objects of class building. But a building may be a role of a heap of bricks and concrete. Inevitably, we reach the conclusion that the perspective of the observer over time must help determine roles and classes.

Existing modeling methodologies force implicit assumptions about roles (adapted to data models as early as in [11]) and classes, implying that analysts have to find the right objects once and for all, and any two good analysts should come up with the same model. Many researchers (e.g. Hirschheim and Klein) have argued that such positions are inappropriate in many systems which demand a flexibility in interpretation of observations, but these opinions have not been reflected directly in information systems design.

However, the idea that there is often a need to change and manipulate representations has been expressed in some limited senses. Schema integration, evolution and relativization are therefore surveyed broadly in the subsequent sections.

2.5.1 Schema Evolution

Schema evolution studies ways to change a schema, without relation to the drivers which may necessitate such changes. The existing work in schema evolution is therefore limited to the definition of operators such as "add a class at X" or "merge subclasses of X" (see [35] and [169], for instance, for a discussion of the schema constraints to be preserved under change). These operators highlight some relevant issues related to maintenance of semantic integrity for inheritance and other relationships under change. But schema evolution is limited:

- It does not allow reasoning about the changes that might improve a sub-optimal schema, discounting the availability of statistics and the user's requirements in aid of this reasoning.
- It is tied to particular data models through its objective of preserving specific constraints.
- Even within the context of specific data models, they impose strong assumptions like "information constancy" (i.e. a schema change must not add to or remove integrity constraints) and "unique inheritance superclass" (i.e. inherit from a unique superclass).

These assumptions are sometimes too restrictive in real life problems.
2.5.1.1 **Seminal Research**

The seminal paper in this area is [12], an proposal motivated by the 20 most common schema change requests observed in the ORION. The authors propose a set of primitives which are complete with respect to a particular graph transformation formalism, and preserve a set of ORION-specific constraints (such as distinct name invariant), which are proposed as reasonable constraints for any object-oriented system. The preservation of these constraints is sometimes guaranteed by postulating arbitrary conflict resolution rules when a constraint could be violated. Though seminal, the proposals in this paper are particular to the ORION model. Some other researchers, such as Zicari, have enhanced [12] and clarified some semantic issues by differentiating structural from behavioral consistency. Formal definitions of consistency are derived and a dependency graph method for propagating consistency throughout the affected parts of the schema is presented. Tanaka et al [157] define a similar system, coining the term "schema virtualization" for the process of creating virtual schemas by defining new classes, relating existing classes, hiding parts of the structure, etc. This work is significant because it suggests the possibility of defining these notions independently of the data models. None of the proposals, however, consider the problem of reasoning with schema in response to changing users' needs.

2.5.1.2 **Recent and Relevant Research**

The most valuable recent research program in this author's knowledge is documented in [124]. This effort could be classified as a schema evolution or integration framework, in that the authors define operators to transfer one schema to another, while preserving the expressibility. This work extends [78] in that it not only allows schemas to be defined through a metamodel, but also defines operators to transform schemas. Schema transformation is not related to any increase in its usefulness or practicality. This framework lacks the meta-properties on the basis of which the goodness of a schema with respect to another can be decided, and as in [78], neither the primitives, nor the model-inherent and domain-inherent constraints are representable or analyzable in formal terms. The valuable insights offered by this database schema manipulation system are:

- Their method based on specialization and degeneration, of modeling hidden parts of a data model, such as the implicit notion of time in Petri Nets.
Their recognition of the model-domain duality in object-oriented models, whereby parts of the domain semantics are incorporated into the schema through 'system-defined' classes.

One effort coincidental with our own is [159], which defines an explicit meta-model for data models. Tresch defines a small number of functions defined on each of his meta-types. Actual schemas are built by instances of the meta-classes. Tresch's meta-model is specific to object-oriented models, as indicated by functions such as "inherited attributes" defined on the meta-type Class. Though it proposes relationships of schema change, this model does not visualize a series of changes or states as a first-class entity. This is initiated by [95] through the notion of schema versions, which introduces (our) notion of history in a limited sense. If a schema evolves by dropping an attribute A at time t, then A should be visible to transactions before t, not to those after t. Thereafter, the notions of version scope, version derivation hierarchies, and creator version are intuitively defined. Kim's proposal is a significant start, but is very specific, as the name implies, to versions. For example, combining parts of more than one version into a new schema, and more than one co-existing version of the same schema are disallowed. Co-existing schemas appear in a concept paper by [123] and in [137] and [6]. None of the schemes reasons with the utility of schemas and versions.

Somewhat tangentially, for software systems evolution, [156] propose a framework to specify the possible configurations which software modules may take over a time. The relationships between these different configurations are specified at design time; the evolution system can ensure that the changes are automatic and consist (and lazy). Schemes such as this are really object migration schemes (as is [114]), since the classes which are instantiated are fully pre-specified. Elements of schema evolution permeate the work because taken as a whole, a gradual change in all instantiations from, for example, a class of Pascal routines to a class of routines in C is defined.

2.5.2 Schema Relativization

One of the earliest schemes meant to manipulate database schema to match the needs of users appears in [50]. In it, Corella considers every entity to be an intension, allowing the relativization of schema and instance. Operators are defined for rearranging generalization hierarchies according to the perspective of a user. While seminal, Corella's work does not define a framework to reason with schema changes, or to relate a user's needs to the goodness of
schemas for that user, in the manner suggested by [65]. Other general schemes to relativize schema to dynamic changes in the user's requirements from the information appear in [112], [120] and [141]. The emphasis in these results is on incremental and ad-hoc changes to traversal paths or groupings in response to changes in data distribution or query type.

[150] and [97] have proposed the tailoring of representations for planning, which is formulated as a traversal from an initial state to a final state through a set of intermediate states using operators. The key concept of interest is that not all planning is done at the same level of abstraction. Initial planning may be done at a high level of abstraction, and depending on need for detail, different abstraction hierarchies may be defined. The essential insights from this work transcend their origins in planning domains, and correspond to our own views:

- Under linearizability and decomposability, objects at a higher level of abstraction preserve the integrity constraints which are present in their most detailed form. This observation is true of schemas as well.
- The act of evolving a schema to match a user's requirements, or the constraints of another schema with which it is to be integrated, can be viewed as a traversal from one state to another.

The productions of the grammar as used in this thesis are analogous of the operators in the planning domain.

2.5.2.1 Specific Types of Answers

Relativization of answers has been studied in a database context in [103] and more concretely in [41][46][47]. The latter suggest the construction of 'abstract pattern classes' which define categories that are most useful with respect to the level of abstraction at which queries are posed. These approaches define good abstractions in the context of a given set of queries in terms of the size of answer sets. The methods are also rudimentary - tables of term correspondences at different levels of abstraction are used to map queries onto higher or lower levels of abstraction. In related work, the authors attempt to define an approximate answer for a query in terms of a pair of answers, \(<C,P>\), where C is a superset of the correct answer and P a subset of it, C and P being easier to evaluate. From the point of view of size of answers, [76] proposes storing prioritized partitions of relations which restrict the size of the answer set. Methods are defined to arrive at an answer which is optimal in size through progressive refinement based on joining and dissociating the partitions. Alternative measures of goodness may be based on the confidence in the sources from which answers are derived.
2.5.3 Schema Integration

Schema integration addresses issues in providing a consistent and coherent interface to two schemas accessed through the query and update facilities of one of the schemas. The issues include integration of different data models in general, integration of two specific data models, and two axiomatizations of a single domain in the same data model. Our concern is with reasoning with elements of schema as they change and adapt to users’ needs. The main contribution of this body of work to is a categorization of the constraints found in disparate models and some transformational primitives. Heiler [77] provides an insightful categorization of heterogenous databases based on what aspects may be visible to a user of the system, and what aspects are transparent, such as:

- Data Model transparency - the data model of the database which contains the queried information need not be known in order to formulate a valid query
- Schema transparency - choices (such as whether an entity is represented as a single database unit or as multiple connected units) made while axiomatizing the domain should not affect the query formulation.
- Time/Context transparency - The time of a query, or the truth/falsehood of other propositions at the time of the query, must not affect its formulation.
- Dependency/Constraint transparency - the constraints and dependencies which a user may see should pertain only to the database which the user is using, not to the databases where the data resides.
- Source transparency - the source of the query should not affect it.

Of interest to us are the first two transparencies; the others are either not relevant to temporal models or eliminated due to complexity. Within these limits reside federated, schema integrated and interoperable databases. Two surveys, [14] and [148], and two journal issues have recently surveyed interoperable databases (see [139] and [148]). The first survey (Batini et. al.) allows a concept to be axiomatized in different ways in different databases. For example, direct relationships in one axiomatization could be indirect relationships in another database, and may be objectified in a third. This study proposes a set of inter-schema properties which relate objects in schema A with their counterparts in schema B. The authors identify algorithmic specification of schema transformations, formal specification of constraint resolution and integration to be significant open problems. Orthogonally, [139] and [36] evaluate
data models as schema integration languages based on expressiveness, recommending an
object-oriented framework.

2.5.3.1 Restrictions on Schema Integration

Schema integration has been defined to imply that each of a set of users of different data-
bases may use a different schema, with mediation being performed only to the extent needed.
Since it is difficult to unify two arbitrary sets of schema constraints, most proposals address a
restricted problem, mostly using logic as an intermediate language. Nevertheless, such
suggestions offer little by way of constructing systems or defining legal transforms.

Most designs customized to specific data models are restrictive. For example, [113]
proposes primitives for integrating databases which are instances of the relational model and
where the semantics of a relation is encapsulated in its signature (i.e. \( R(a,b) \) and \( S(a,b) \) are
different since \( R \neq S \)). Once a virtual database is defined using the proposed meet, join, fold
and other operators, a virtual database query can be automatically unfolded to the component
databases. Iwaihara et al [84] investigate integration methods for nested relational and
network databases, addressing a similarly restricted problem.

Krishnamurthy et al [100][101] address schema discrepancy problems at a level which
we are interested in, where the differences transcend reconciliation using relational-like oper-
ators. The authors acknowledge that in any domain, a given entity may be viewed by different
users as an attribute value, an entity or even a class with its own instances. They develop a
high-level language with facilities to define relations, entities and attributes from initially
undifferentiated schema elements related by statements like Associated\((a,b,c)\) where 'Associ-
ated' is a system-defined higher-order relationship among schema elements, much in the
spirit of our earlier work in [70]. Jeusfeld and Jarke [89] also propose a similar scheme, for
different reasons. We agree substantially with this perspective on the nature of schemas and
define a similar functionality in our framework, but we extend the functionality with the
ability to reason about and trigger schema updates based on conditions.

Bertino, in [21], develops an general object-oriented system to integrate different data-
bases. Bertino assumes that a global integrated schema could be derived from the component
schemas using two kinds of mappings: Operational Mapping and Structural Mapping. The
proposal is a simple translation using triples of local name, global name and mapping scheme.
An object-oriented query consists of a target clause, a range clause and a qualification clause.
The target clause of a query may be specified using a conjunction of literals such as (Permanent \& Student) (x) (i.e. system-defined formulae are usable as names). Apart from the same shortcomings as [100], Bertino's proposal does not address cases where the component classes cannot be defined as conjunctions of literals of existing classes, as discussed in [100].

2.5.3.2 Federated Database Models
Federated databases investigate common global models for heterogeneous databases. Where this involves different data models, the integrating system must be capable of reasoning with model-inherent constraints. Much of the early research in federation attempted to define a single super-data-model covering most other models, and usable as a medium for translation. Demurjjan's MLDBS [54] defines a Multi-Lingual Data Base System with a kernel which stores atomic attribute-oriented information. It supports data model and language transformations through an interface for each language which translates to and from the kernel language. Each feature of a particular data model is transformed into the features of the kernel model in terms of Basic and Acquired Transformational Keywords. The former represent the terms native to the Kernel model, while the latter are used to capture the implicit semantics of the source models. Similarity of data models is defined on the basis of the number and types of transformational keywords needed to describe them. For example, the relational model may be seen as a collection of two types of elements: Relations without keys (weak entity sets), and Relations with keys. The latter is defined by Reln(Tclass2) = \{(type-name),(ident-keyword),(elem)^*\} to capture the fact that such a relation has a name, a distinguished part that serves as identity, and 0 or more other elements. The proposal is valuable, though it lacks representability for such important relationships as functional dependency. The problems involved in maintaining such constraints in the context of updates is introduced in [158] and [143] for deductive and object-oriented databases.

In the direction of enriched meta-schemes, [131] is an early advocate for representing the semantics of schema primitives explicitly, in order to be able to reason with them under change. Subsequent proposals such as ViewSystem [93] present comprehensive frameworks including a minimal meta-class hierarchy consisting of extensional classes, external intensional classes, and derived intensional classes. The primary insight provided by ViewSystem lies in the VODAK scheme's ability to lazily materialize newly created classes to minimize on-line time cost, an idea exploited well in by Tan and Katayama. A more comprehensive framework to represent schema constituents and constraints is found in [78], where Hong
and Maryanski propose a meta-model for the object types, relationship types and constraints among objects related by instances of the relationship types. This is a useful basis for explicating schemas, but is unable to reason with constraints due to its non-formal descriptive language. Another noteworthy formalism in this category is [159]. There is a body of research on reasoning with constraints and their use in integrity maintenance, represented by [92], [166] and [85]. The main thrust of Karadimce and Urban [92] is to identify from the structure of constraints expressed in an F-Logic like language their dependencies upon each other and their effects on insert, delete or update to the database.

2.5.4 Implementation Issues

Two large implementations, Telos and DAIDA, exist. Telos [115] is based on the premise that a variety of knowledge needs to be explicitly represented and reasoned with in information systems development. The knowledge includes application knowledge, user models, system requirements, versions, teams, methodologies, and an explicit representation of time.

Telos is a software requirements specification language covering the points above, and has been used to implement DAIDA. It includes a metamodel which recognizes that attributes, and (concrete and abstract) objects should be treated similarly. Thus, for example, John, age, and Person are all treated as instances of the same kind, proposition, a first class object. Hence the domain, range and name of 'age' can be queried just the same way as the name address or age of 'John' can be queried, which indicates a higher order capability. Telos adds a temporal dimension to inferences, supported by the transitivity, instantiation and specialization axioms for time. The usability of these features in changing schema is limited because appropriate meta-attributes which refer to the schema are not defined within Telos.

DAIDA [86] uses the Telos language to address issues in evolving information systems. It implements a system to explicitly represent the requirements modeling, design and implementation process for a project, along with the factors which influence each step. This aids the transition from requirements, through design, to implementation. Daida is designed as a passive informal aid to the software developer, to be used as an off-line index to the history and dependencies in a software product. This contrasts with the aims of adaptive information systems, which are active, on-line monitors of changing events, information, requirements and schema, with the capability to effect changes that increase schemas' utility. The DAIDA model aims to satisfice a set of nonfunctional requirements (e.g. cost, time) through goal analysis,
decomposition and requirement-specific methods for goal reconciliation, hinting at the applicability of utility measures to such problems. Telos and Daida thus point towards the applicability of goal-driven analysis for guiding change and for automated support systems design tasks. They do not propose means for on-line modeling or reasoning with schema.

Active databases (e.g. [Dayal]) research issues in monitoring a database for particular conditions triggered by events, and initiating actions at that point. Initiating and executing object or schema migrations triggered by incremental updates may involve some ideas from active database, but these are implementation issues subsequent to conceptual modeling.

2.5.5 Logics, Structures and Constraints

Increasingly, researchers (e.g. [49], [66] and [9]) are formalizing complex schemas as graphs, with the resulting ability to define operators and algebras operating on lists, sets and aggregates through a common medium (since these can be represented as graphs). In some sense, we adopt and extend this perspective in our proposal through the definition of objects and schemas via grammars (which generate derivation trees and graphs). Grammar-defined graphs allow constraints to be placed on the structures of the graphs. The constraints themselves need a language to express and reason within, which is often logic-based.

Reasoning with and manipulating schema involves questions such as "What function may can map schema elements A and B into schema elements C and D?". I.e., we ask:

\[ C(x,y) :- f(A(p,q,r), B(u,v)) \]
\[ D(x,y,z) :- g(A(p,q,r), B(u,v)) \]

such that conditions \(P,Q\) hold of \(C\) and \(D\), where \(f\) and \(g\) are function variables. This constitutes second order reasoning, since function variables are involved. If no restrictions apply, this is unacceptable because resolution in higher order logics is undecidable. The trade-off between the expressive power and computational complexity of logical languages is well represented in [104], and in the specific case of terminological reasoning, which is a case of reasoning with unrestricted schemas, the problems are exemplified by [119] and [140]. A review of the subject would situate our investigations.

Propositional logics (see [29] for definitions and relevant results) are weak in that statements can not be quantified over members of the domain of discourse. Truth in a propositional logic is decidable, but takes exponential time in the size of the statement. First-order logic (FOL), which allows quantification over elements in the domain, but not over relations or
functions, is semi-decidable - true statements can be proven in exponential time, but false statements may not be proven false. Some decidable but weak FOL variants have been proposed. Second order logic is strictly needed to express statements about a domain only if statements need to be made about all possible relationships or functions. Since uncountably infinite number of functions exist on countably infinite domains, decision procedures for higher order logics in general can not exist (see, for instance, [8]). Some proof procedures (to decide truth, but not falsehood) have been proposed which preserve the higher-order nature of the logic (i.e. no restriction on function and relation types, see [81][64]). However, the restrictions which they place on the unification procedure or results are hard to justify intuitively, and in fact, admit unpredictable effects.

There have been, however, two major interpretations of second order semantics. The standard model interprets the function variables and relation variables extensionally, so that the domain of possible functions is uncountable. The other major theory is the general Henkin model of second order logic, which interprets function and relation variables intensionally. In this case, second order logic is second order only in syntax. Semantically, the function and relation variables are interpreted over countably finite sets of functions and relations, respectively. Under this semantics, second order logic becomes equivalent to a many-sorted first-order logic, which in turn is translatable to a one-sorted first-order logic. In other words, the general Henkin model allows us to add meta-structural syntactic sugar to first order logic (which is useful for purposes of formalizing a theory), but has no real expressive advantage. Recent proposals for systems along this line include [1], and in terms of languages based on these concepts, lambda-Prolog [116][110], C-Logic [45], F-Logic [94] and HiLog [42][44]. Each of these has different specialities within the semantic constraints of the Henkin model.

Are true higher-order features, or only meta-structural features, required to reason practically with schema? Alternatively, when transforming one schema to another, are all possible functional transformation of their elements allowable? We choose to answer this question in the negative, both for practical and intuitively justified reasons. Database administrators [71][72] would normally limit the ability to transform schema quite severely. It is reasonable to assume that a limited number of functions will be allowed to unify one schema into another, and therefore, to consider second order logics with first order semantics as a vehicle to express and reason with dynamic object and schema semantics. The use of such meta-level logics have been investigated in, for example [30], [40], where a higher level (which refers to first-order
predicates and functions as objects in their domain of discourse) is used to control some inference process of logic programs or databases.

On the subject of domain-specific logical semantics, the more general proposals (excluding specific proposals related to semantics of colors etc., as in [142]) include the LOGIN system [4], which builds inheritance into the lattice structure of its schema, rather than deducing inherited properties. LOGIN uses the union-find algorithm to reduce the height of the lattice in order to reduce the amortized cost of inheritance reasoning. More comprehensive schemes to integrate two systems of inference include results such as [16][17], which explore schemes to integrate logic programming with functional programming in a single system. The motivation for these efforts is to execute logic programs at least partially in a directional way (with fixed inputs and outputs) rather than as bidirectional declarative programs, to increase efficiency. A theoretically firm basis to this research is provided through the concept of E-Unification, which replaces the unification procedure of logic programming with unification based on equality of terms as defined in an equational (functional) programming language. An alternative approach to unify the two paradigms is narrowing semantics, which refers to the minimum unification-based substitution required to be able to reduce a term. In this thesis, hybrid reasoning appears in the form of temporal reasoning in a grammatical framework which "calls" an FOL engine to do conventional logical reasoning.

Intensional logics [164] are significant because of the temporal reasoning necessary in object migration and schema change. However, whereas qualitative definitions of model preference (models equate to abstractions in our framework) are favored in AI, our proposal limit ourselves to basic quantitative measures of goodness of models as in utility theory [62].

Another influential program is constraint analysis, the interaction of constraints in dynamic systems. Urban and Delcambre [161][162] propose Generalized Constraint Analysis, where model-inherent and external constraints can be expressed through a combination of a graphical language and an F-Logic-like language [95]. [161],[162], [53] and [92] all present axioms which are used to determine the dynamic conditions (insertion, deletion etc.) under which constraints of a particular form could be violated. A syntactic categorization of such constraints into "object-centered" and global constraints is given, which may be useful as a basis for determining the effect of an incremental update on objects or schema. Navathe [118] similarly identifies objects as being "fully" and "partially" internally identified if their identity is not (is, respectively) dependent on the existence of other objects. These studies are valuable
in prior analysis of systems that may be modeled using our framework, but not directly relevant.

Similarly, Constraint Logic Programming (as in [Dincbas, Simonis, Hentenryck], [Wilson, Borning], [Heintze, Michaylov, Stuckey, Yap]) has little direct relevance to dynamic and adaptive databases since search space is not at issue, and reasoning with schema constraints is more symbolic and higher order than numeric, where these techniques are most useful.

2.5.6 Peripheral Topics

We mention some areas of peripheral interest to this thesis for the sake of completeness. There is a need to discover existing schema and object semantics before change patterns can be specified. Schema extraction addresses tools to discover multiple schemas so that administrators may specify interconnections and dependencies with respect to them (as in [147]). Similarly, data mining and knowledge discovery address problems in learning from extensional data. Our proposal concerns prior legal change relationships, as opposed to discovered relationships, making these efforts peripheral to our interest.

Intension subsumption in schema integration involves name resolution. [55],[33] and [43] have proposed approaches to equating terms on the basis of personal construct theory, or synonym, antonym, homonym and generalization relationships. These are in the context of natural language and query reformulation, both peripheral to our research.

User modelling in artificial intelligence aims to represent agents with beliefs, knowledge, goals and capabilities. These may be used to motivate communication choices through presuppositions and implicatures. Though we propose that representation should change as the users' needs change, we avoid the derivation of users' needs and therefore, user modeling as a subject. For answering natural language queries, Hobbs and others have suggested techniques to isolate the users' presupposed vocabulary. This is logically completed by adding other terms needed to derive logical dependencies among the presupposed terms. Such work applies to richly connected domains, not to the sparse domain of databases. An early position on utility is that of Russell et. al., who stress the pragmatic coherent view that an answer is good if it promotes the well being of the questioner, but to them, goodness relates only to time-cost. Doyle et. al. prove that a globally acceptable schema for a community of users is, in general, impossible. These results motivate and bound our work, but do not affect it directly.
2.5.7 Recent Results

It is interesting to see recent published results separating structural from semantic constraints and recommending a minimal commitment semantic model, as in [129] [111], to allow schemas to be mediated easily. The growing importance of reasoning with processes is shown in recent work as well [146]. The idea of object-behavior diagrams has also been made "active" in a way similar to our proposals in [23]. Finally, work is continuing towards enhancing temporal indices in ways that would address our concerns in [87]. The continuing work in these areas indicates to us that there is sufficient interest in the topics addressed in this thesis to continue work on these topics towards innovative results in the near future.

2.6 Summary

- Temporal data models typically define algebraic languages on conceptual time intervals or points, not constraints among successive attribute values, class memberships, or changes to class definitions. Temporal databases often address implementation through RDBMSs.

- Object migration or schema change models are often unable to reason with changes. They often suffer from a lack of expressive adequacy and extensibility of their primitives. As well, systemic shortcomings such as a lack of class-centered modeling versus process-centered approaches, unified static/dynamic modeling scheme, encapsulated (not distributed) dynamic schema, and analyzability cause problems.

- Schema change formalisms typically neglect two vital real-life concerns: Correlation of user requirements and optimal schema, and freedom from a pre-conceived ontology.

This thesis develops a framework which addresses in large part the shortcomings above. The emphasis is on the relatively easier problem of object migration, though our research yields interesting implications for schema change as well.
Chapter 3  Framework Architecture

We address the problems introduced in section 2.6 by proposing a unified framework for static and dynamic relationships among objects and elements of schema. In our framework, recognizable objects and collections are inter-related through directed semantic constraints. Related entities may be concurrent in time, or in the relative past or future. These relationships define a network of constraints. Certain sub-structures of this network are interpreted as object-states at particular points in time, while other sub-structures can be interpreted as sequences or patterns of object-states over a period of time.

Figure 3.1 illustrates these concepts. The thick lines denote temporal derivations, while the dotted lines denote the resulting temporal sequence of states, implying that state (class membership) E follows state D in time. The thin lines denote static relationships (say aggregation). If an object changes from D to E, the change may not affect the aggregated object's membership in class A if class B also happens to be a superclass of classes D and E along the inheritance hierarchy. However, if a subsequent change in state to another state F occurs, the aggregate object may migrate from class A to another class A'. Such dynamic constraints are representable and amenable to reasoning with within the framework defined in this thesis.

![Diagram](image)

**FIGURE 3.1**  Static and Dynamic Relationships Expressed Graphically

Our proposed framework is based on Augmented Transition Networks (ATN, a formulation of attribute grammars, see appendix C) to reason with the static and dynamic relationships between concurrent and consecutive entity-states in time. The justification for this framework is addressed in chapter 6.
3.1 Structure, Essential Modeling Devices and Features

In terms of this grammar-based foundation, we interpret objects and collections evolving through time as structures defined and manipulated by grammars. A dynamic and adaptive information system framework physically stores substructures of graphs such as that illustrated in Figure 3.1 and uses ATN-based algorithms to perform query and update operations on the stored information. The dynamic relationships captured by the grammar may be categorized as follows.

- **Essential Dynamic Devices** are needed to describe all kinds of change. For example
  - Which states temporally precede / succeed / overlap other states.
  - Legal associations among states and objects under change, and the implications that a particular state has on future states of existence of the same or other objects.

- **Dynamic Features** are customizable aspects of change descriptions. Though the customizations may be domain-specific, dynamic features often capture the most significant characteristics of changes in a particular domain. For example:
  - Whether an event necessarily leads to a change in object or schema state.
  - Whether changes are reversible, whether they involve multiple inputs/outputs, whether a change at a time t enables or prevents changes in the future.
  - Whether an object identity is preserved through a change where other identities are destroyed and/or created.

As we show in chapter 5, if the allowable changes in the domain are represented using the essential dynamic devices, the model can be used in conjunction with static information to design better information schemas. The static semantics, as well as those of both essential dynamic devices and features in the resulting schema are interpreted in terms of devices associated with ATN attribute grammars such as:

- **Registers**: inherited or synthesized qualifiers associated with symbols of the grammar. ATN registers are interpreted as object / schema attributes and meta-attributes used to reason with change.

- **Preconditions**: Conditions that may be global, or local to an ATN transition. They determine whether a candidate object migration or schema change can proceed.

- **Utility**: A measure of the usefulness of an object or its representation, is a special meta-attribute in our framework that may be used to trigger or prevent change.
• **Actions:** Modifications to register values indicating updates to object attributes or meta-attributes. Actions accompany migration or schema change, and are not deducible from the antecedent or consequent state definitions in general.

• **Postconditions:** ATN constraints to be maintained after the actions. These are interpreted as conditions that must hold in the future in order for a change to be valid at a certain time.

• **Grammar Type:** Historical contexts within which objects or schemas change are described by sentences in a language specified by the ATN grammar. The categorization of grammars in formal language theory defines the legal temporal relationships among the changing object or schema states.

Once the semantics of the essential devices and features are interpreted in terms of ATN registers, actions, conditions etc., these are manipulated by algorithms adapted from ATNs and data structures adapted from computational geometry, as shown in chapter 6. These algorithms allow us to infer properties of objects from (possibly partial) stored histories of the objects in the same historical context (e.g. plan). Past status and possible future states may also be inferred from these partial histories using the algorithms. Manipulations such as rollback and forget can be implemented on object histories while preserving all dynamic constraints, as also updates to future states.

The same algorithms and data structures can operate on a different set of essential devices and/or features in the case of schema, as indicated in chapter 8. In the case of schema, the devices and features are useful to specify how a schema may change legally to accommodate the needs of a user over time. They can also be used by an administrator in a multi-database environment to specify legal schema change paths for automatic migration of objects across data models, as well as evolution, integration and relativization of schemas.

In general, reasoning with schema is difficult. For example, if schema A is logically represented by the predicate \( a_1(p_1, q_1) \land b_1(r_1, s_1) \) and schema B is logically represented by the predicate \( a_2(p_2, q_2, r_2) \land b_2(s_2) \), \((a_i \text{ and } b_i \text{ are predicate constants, } p_i, q_i, r_i, s_i \text{ are attribute constants})\) schema integration may require the computation of predicate and attribute variables A, B, W, X, Y, and Z such that

\[
A(W,X) \land B(Y,Z) \equiv_s a_1(p_1, q_1) \land b_1(r_1, s_1) \land a_2(p_2, q_2, r_2) \land b_2(s_2)
\]

(Where \( \equiv_s \) denotes semantic equivalence of the intensions of the left-hand-side and right-hand-side expressions). Computationally, this statement implies the discovery of the (most general) unifier for the following:
When this unification involves finding a binding for a function or predicate variable such as \( \textit{A} \), this is a second order unification problem, which is provably undecidable. We propose to use a first order semantics for this essentially higher-order problem, as proposed by previous researchers for other problems, since pragmatic representations of domains and schemas must be finite entities.

### 3.2 High-Level Logical Architecture

We propose the logical architecture shown in Figure 3.2 for our formalism for managing change in domains and schemas.

---

**Figure 3.2** Dynamic and Adaptive Information Systems Architecture
Figure 3.2 illustrates an idealized architecture for an object migration and schema adaptation framework. In implementing this ideal, many issues need to be resolved, some of which are beyond the scope of a single thesis. This thesis defines only the formal backbone for schema and object representation and reasoning. Specifically, some issues which are not addressed include:

- The physical distribution and logical binding of the different blocks in an implementation. The area of heterogenous distributed databases addresses a number of these questions. Frameworks such as Encina and DCE can address these implementation issues.

- Description and manipulation languages for the schema change and object migration modules. Since they involve interfaces to heterogenous systems, such interface issues are normally resolved through standardization, which is not the subject of this thesis.

User modeling systems which may be used to enhance the value of such a framework by storing and reasoning with facts about each user's knowledge and goals. In terms of the figure above, the issues which are addressed in this thesis include:

- A skeletal query language, dynamic domain description language and adaptive schema definition concepts based on a basic alphabet for dynamic domains, and a grammar-based computational framework.

- Concepts of utility/usefulness of objects and schema with respect to a set of requirements.

- Dynamic data structures to facilitate implementation of the conceptual framework.

- Some issues regarding semantic adequacy and performance characteristics of the framework used for object migration and schema adaptation.

3.3 Components of the Dynamic Modeling Framework

Data modeling concerns the creation of effective abstractions which capture significant distinctions and hide irrelevant details. High-level abstractions which are specially useful in a particular domain may hide factors which capture information relevant to other domains. Yet, in order for a framework to unify a diverse set of dynamic domains and schemas, a common level of representation should exist. This base should be broadly acceptable, and extensible to many alternative sets of high-level modeling primitives. Accordingly, we propose a representationally powerful computational framework, called the dynamic founda-
tion (D'FOUNDATION, for brevity), based on formal language grammars, which underpins the computational aspects of all the static and dynamic semantics of the domain.

Whatever the computational model may be, a user-interface language is needed for the specification of dynamic behavior, for updates and for queries. We conceive a two-layered language, reflecting the need for a basic vocabulary and an enhanced layer capturing domain-dependent refinements.

- The core of the dynamic language (D'LANGUAGE for short) is a small set of essential dynamic devices called the D'ALPHABET. These are universal primitives for dynamics.
- D'Alphabet is enhanced by an extensible language known as D'DIALECT (the dynamic dialect). This language specializes d'alpha for objects or schema, and also captures both the essential devices the customized features of specific domains.
- Each component of d'alpha and d'dialect maps onto specific constructs of d'foundation. Hence the trade-offs between expressive content and computational complexity are easily derived.

![Figure 3.3: Structure of the Proposed Modeling Framework](image-url)

The layers of d'language as shown in Figure 3.3 realize the logical architecture of Figure 3.2 yielding these benefits:

- Intuitive correspondence between the semantics of the modeled domain and the vocabulary of d'dialect.
- The ability to apply a common informational model to schemas and objects through d'alpha.
- The ability to unify static and dynamic computations involving objects and schema identically in d'foundation.
The fact that the lower layers are common to both object dynamics and schema change frameworks enables object/schema relativity, the notion that the entities which one user queries and manipulates can be the templates in terms of which another user defines her domain [50]. This concurs with some researchers' suggestions [159] that operations on objects are semantically meaningful to schema as well and vice versa. However, we do not agree with the need for a universal algebra for objects and schema, due to the reasons given in [78]: the same operation when applied to an object entity, and a schema entity, may imply different consequences.

Two approaches may be taken to capture the differences between object and schema:

1. Propose a single language and situate the semantic differences among the procedural semantics of objects, and objects that happen to be schemas.
2. Propose separate high-level operators for objects and schema, and a mapping to a common computational base.

We adopt the second approach, and differentiate between dynamic object dialect and the dynamic schema dialect, D'DIALECT(O) and D'DIALECT(S). D'foundation and d'alpha-letter are common for objects and schema because at the computational level, the primitives do not encapsulate semantics particular to objects or schema.

We illustrate these dialects and their value in modeling useful dynamic situations in the subsequent chapters. Apart from the architecture, we describe the method of constructing dialects correctly upon d'foundation. We first investigate object dynamics (as distinguished from schema dynamics), the study of, and reasoning about, objects under change, through a study of d'alpha-letter and d'dialect(o) in section 3.4 and chapter 4 respectively. We postpone discussion of the computational mechanism d'foundation until chapter 6. Subsequently, we build upon d'alpha-letter and d'foundation to define d'dialect(s) in chapter 8.

### 3.4 D'ALPHABET - Core Terminology for Dynamic Domains

The basic informational model underlying both object and schema definitions are included in d'alpha-letter. We emphasize the fact that our original contribution in this chapter is restricted to the modeling construct called reaction, and less importantly, to the notions of value-sets and other supporting dynamic terminology. The remainder of the definitions,
including intervals, objects, and relationships such as overlap are derived from previous work, but presented here because of their importance to our subsequent discussion.

Our description of static scenarios of complex objects are founded on identities and directed relationships. This terminology derives set (and class) memberships, aggregations and lists, and also some founding structures of dynamic representations. Concepts used to describe dynamics of entity existence include notions of time intervals with start and end points, before and after relationships, legality, deep and shallow associations of entities and utility (or usefulness) of entities. These concepts lead to the notion of reaction in d’alphabet. We start with some definitions:

**Definition 3.1** An object is a part of a given domain of discourse which, from the point of view of an observer, embodies properties of relevance to the observer.

1. Objects stand in one-to-one correspondence with identities.
2. Objects participate in relationships with other objects.

The concept of identity has been contentious, as evidenced by the philosophical debate on the nature of empirical evidence and structure of knowledge (appendix A). In order for different observers to classify objects appropriately (and more basically, to attribute properties to appropriate objects) from their perspective, we need the notion of identities independent of specific intensional or extensional entities. Therefore we define:

**Definition 3.2** An identity is a distinct and distinguishable index unconstrained by limits on co-location in space, time or any other dimension with a different identity.

**Notation:** We use the terminology \( \text{Id(object)} \) to refer to the identity of an object.

An identity is a distinct placeholder from an observer’s perspective to which different aspects (such as space, time, or other attributes) can be attached. The definitions above imply that two objects (distinct individuals) could be indistinguishable with respect to the measurable attributes associated with them as projected on a particular observational plane, including with respect to their habitation in space and time, yet be distinguishable as distinct objects capable of distinct reference. Distinct objects must therefore be distinguishable, but this distinguishability may not be manifested through some measurable dimension in a particular plane of observation. According to the definitions above, individuals, aggregates and
groups are bonafide objects. A group of objects could be a distinct object (a set object) in its own right. It could be two distinct objects, if the set is usable for two different purposes.

Example 3.1 If Persons are objects, the group of persons \{John, Mary\}, can denote a family (be a Family object, Jones, in that role). If the same objects John and Mary operate together as a team, \{John, Mary\} could act as a team (be a Team object, Firestone, in that role). Conversely, the concept of the Jones family may extend only to the financial assets of John and Mary, omitting all other aspects, for taxation purposes.

3.4.1 Attributes

Identities, and therefore objects, convey little information by themselves. Their primary reason for being is to act as references about which information of interest to an observer may be associated. Attributes are the medium for capturing the information of interest in d’alphabet.

Definition 3.3 An attribute \(a\) is a named directed relationship (function) from a class \(x\) to another class \(y\), written \(a : x \rightarrow y\).

Notation: For instances \(x\) and \(y\), we say \(a(x) = y\), where \(y\) may be a collection of objects.

A collection is a grouping of the objects in a domain and may itself be an object. A collection may group objects according to some measurable criteria determined by their attributes, or arbitrarily, according to no measurable criteria other than the objects' (post-facto) grouping into a single collection. The objects thus grouped are called the members of the collection. Collections exist in order to restrict the domain for the purpose of querying, updating, and maintaining integrity. \(C \in \text{Collections}(A)\) (equivalently, \(A \in \text{Member}_\text{of}(C)\)) is valid of an attribute of the object \(A\), post-facto to its inclusion in collection \(C\).

Definition 3.4 A collection \(C\) is an intensional collection if the members of \(C\) belong to \(C\) because they satisfy some conjunctive condition based on their measurable attributes.

Definition 3.5 A collection \(C\) is an extensional collection if it comprises objects which are in \(C\) for a reason external to their measurable attributes (or because they are in \(C\) because they satisfy only a disjunctive condition based on their measurable attributes).

Our concern in the domain of objects (not schema) lies mainly, but not exclusively, with intensional collections which we call classes, and changes to their memberships. We define the following basic attribute types with regard to intensional collections.
Definition 3.6  An attribute A is a *definitional attribute* with respect to an intensional collection C if and only if every potential member O of C belongs to C by virtue of the fact, among other conditions satisfied, that either:

1. Necessarily A:O -> 0 (A is necessarily undefined on O); or
2. A(O) = P (A is defined on O), and \(((\lambda A ) f) (P)\) is true, where f is some conjunctive condition on A.

Each of these characteristics is referred to as a condition associated with the definitional attribute A.

**Notation:** If X→ is the vector of definitional attributes with respect to an intensional collection C, we use the terminology Defn(C) = f to refer to the logical wff of the definition of C if ((λX→) f) is the conjunction of the conditions associated with the elements of X→. Defn_Attr(C) refers to the set of definitional attributes of C. The notation ((λX) f) is adopted from lambda calculus (as in [110]) to denote the logical proposition f as a function under abstraction of the variable (attribute) X.

Intensional collections are determined by objects which satisfy a logical definition in terms of predicates on their attributes. Definitional attributes are attributes which are qualified by some predicate in the definition.

Example 3.2  In the case of objects, the attribute *Driver's Licence Number* may be definitional with respect to both the class of NonDriver (condition 1 above) in that non-drivers necessarily do not have drivers' licences, and Driver (condition 2 above), in that drivers necessarily do, and the number is restricted to the year of application.

Definition 3.7  An attribute A(C) -> Y is a *factual attribute* of C if there are no constraints on x due to the fact that C ∈ Collections(y), where A(y) = x, such that x may be either 0 or any value in range(Y).

**Notation:** Fact_Attr(x) = y refers to the set of factual attributes, y, of collection x.

Thus, definitional attributes are either obligatorily present or obligatorily absent for instances of the classificatory grouping, and factual attributes are optional.

### 3.4.1.1  Distinguished Attributes - Utility

An identity by itself is only an index into a set of properties which the observer deems useful to treat as a single unit. In a flexible domain representation, each object exists due to its utility to an observer as a focus for a set of properties. Utility itself is not visible as an attribute but acts as a meta-attribute; it may be either a quantitative measure over a continuous scale, or a discrete 0/1 value.
Definition 3.8 Utility is a distinguished attribute which measures the value of the existence of a particular object-state with respect to a set of observer-specific criteria.

Notation: Utility(x, C, Time) refers to the utility of an object x as a member of a collection C at time Time.

If, due to changes in the domain or the observer-specific criteria, the utility of an object drops below some threshold, the existence of that object-state may be questioned. Since representations (schemas) are themselves objects, this constitutes the basis for change of representation. Objects, Identities, definitional and factual attributes and utility are thus the primary concepts involved in the static definition of configurable or flexible domains.

3.4.1.2 Distinguished Attributes - Identity Bindings

The second meta-attribute recognized as special within our framework is binding, which is defined on identities.

Definition 3.9 A tight binding is a function f:A -> B which relates the identity of B to the identity of A. B is tightly bound to A implies Id(B) is undefined if Id(A) is undefined. A loose binding is the converse of a tight binding.

The definition of tight binding based on identities implies that the object B is existence-dependent on the object A; B ceases to exist if A ceases to exist. This is not true for loose bindings. By referring to the commonly understood properties of functions, all attributes of an object O are undefined if Id(O) is undefined. Tight bindings are realized by constructors, while loose bindings are realized by evaluators (section 4.3.1). A similar concept in relational database systems, is that of strong and weak entities. Bindings and attributes together derive the different static object-oriented relationship types.

3.4.2 D'Alphabet for Dynamic Descriptions

Corresponding to static definitions, a basic alphabet for discussing change is required. This vocabulary is based on prior notions of time intervals, before, after, overlap and other relationships between time intervals, a simplified notion of legality, and ultimately, a derived notion of reactions. This dynamic alphabet helps to derive usable dynamic primitives relating changing objects as well as changing representations.

The conventional abstraction of real time is a dense, two-way infinite total order isomorphic to the set of real numbers. In finite representations, time is commonly approximated.
as a sparse two-way bounded total order isomorphic to a contiguous sequence in the set of positive integers. Due to the ease in performing interval arithmetic [5], our temporal ontology is founded on half-open intervals. Unions, intersections, subtractions and additions of half-open intervals yield other half-open intervals, in contrast to the case with closed intervals, for instance, where [1,10] - [2,3] yields asymmetric half-open intervals.

**Definition 3.10** Time intervals, denoted \([t_1, t_2)\), \(t_1 \leq t \leq t_2\), represent all time \(t\) such that \(t_1 \leq t < t_2\). \(t_1\) is the starting point of the interval, and \(t_2\) is the ending point. The interval \([t_1, t_1')\) (see note below) represents the time point \(t_1\). A time point \(t\) lies in an interval \([t_1, t_2)\) iff \(t_1 \leq t < t_2\).

Note: If \(t_1\) signifies the time point isomorphic to integer \(i\) in a sparse domain, then \(t_1'\) signifies the "next" time point mapped isomorphically to integer \(i+1\). In a sparse domain, therefore, a time point \(t_1\) represents interval \([t_1, t_1')\), or the closed domain \([t_1, t_1]\).

**Notation:** We refer to \(\text{Start}(I)\) and \(\text{End}(I)\) as the starting and ending points of an interval \(I\).

We assume that a granularity of the time line suitable to the application, with concatenation and intersection operators as usually understood in set theory. Given objects (identities) and attributes as defined earlier, we now state that every attribute value associated with any object is also associated with a time interval.

**Definition 3.11** A value interval associated with an object \(id\) and an attribute \(a\), denoted by \(<id, a, v, [t_1, t_2])\), signifies that \(a(id) = v\) at every \(t\) in \([t_1, t_2)\). A value interval set (value set, in short) denoted by \(<id, a, \{<v, [t_1, t_2])\}\) is a set of value intervals.

**Definition 3.12** An object \(id\)'s attribute value set, written as \(<id, \{a, \{<v, [t_1, t_2])\}\}\), is a set of value sets for \(id\) and every attribute \(a\) of \(id\). It is synonymous with a set of statements to the effect that \(a(id) = v\) during the time interval \([t_1, t_2]\) for every \(a\) in the set and every value set thereof.

We can thus refer to \(id\)'s attribute value set as a whole, or to a value set with respect to a particular attribute \(a\).

**Notation:** We write \(a(id) = v\) \([t_1, t_2]\) to denote \(<id, a, v, [t_1, t_2])\) when convenient. \(\text{Value}(T)\) refers to \(v\) in a value interval \(T = <id, a, v, [t_1, t_2]>\); \(\text{Interval}(T)\) to refer to \([t_1, t_2]\). \(\text{Val}_a(T'(T'))\) refers to the value set \(<id, a, \{<v, [t_1, t_2])\}\) corresponding to attribute \(a\), if \(T'(T')\) is the attribute value set of \(id\), denoted by \(\text{Att}_a(id)\).

We often ask whether an attribute is currently (at the real time of interest) held by an object. For this purpose, we assume a distinguished time point \(tn\) signifying current time. If
interval(T) = [t₁, tₙ] for some T ∈ value_set(a) (att_val_set(id)), it signifies that the value v has been held for a since t₁ and continues to hold at the current time. We say that the value v is current; if interval(T) ≠ [t₁, tₙ), then value(T) is said to be historical. An attribute is current or historical depending on whether or not its value set contains a current value. We assume a distinguished value Φ, signifying "value unknown". An attribute whose current value is Φ is current.

Researchers in temporal databases have established a terminology for expressing relationships among temporal intervals. We adapt some of the terminology from earlier research (see [63], for example) in our framework.

**Definition 3.13** An interval i₁ = [t₁, t₂) precedes another interval i₂ = [t₃, t₄) iff t₂ ≤ t₃. i₂ is then said to follow i₁.

**Definition 3.14** An interval i₁ = [t₁, t₂) immediately precedes another interval i₂ = [t₃, t₄) iff t₂ = t₃. i₂ is then said to immediately follow i₁.

**Notation:** I_Prec(I, O, A) refers to the interval J immediately preceding I such that A(0) ≤ J ≤ A(0). I_Foll(I, O, A) has an analogous meaning. Prec(I, O, A) and Foll(I, O, A) denote preceding and following intervals.

Immediate precession and following imply precession and following, by definition, while trivially, precession and following are irreflexive, asymmetric and transitive relations.

**Definition 3.15** An interval i₁ = [t₁, t₂) overlaps (sometimes referred to as existential during) another interval i₂ = [t₃, t₄) iff there exists at least one point t that is in both i₁ and i₂.

**Definition 3.16** An interval i₁ = [t₁, t₂) covers (sometimes referred to as universal during) another interval i₂ = [t₃, t₄) iff every point t that lies in i₂ also lies in i₁.

Overlap is a reflexive and symmetric relation. Cover is a reflexive and transitive relation. Trivially, i₁ covers i₂ implies that i₁ overlaps i₂.

**Notation:** Covers(I, O) and Overlaps(I, O) refer to the intervals related to O which cover or overlap (respectively) interval I.

**Definition 3.17** If covers(i₁, i₂) and covers(i₂, i₁), then i₁ = i₂; they are the same interval.

Intervals are objects with two observable attributes, the starting points and ending points. As mentioned earlier, two objects which are observably identical may or may not in
fact be identical. In our framework, the ontological status of intervals is special in that intervals that are the same are in fact, identical. The definitions regarding intervals above hold whether time is assumed to be isomorphic to real numbers or integers. Hence, they are applicable to real time as well as to its finite and sparse representation in computers.

3.5 Legality and Reactions

Dynamic and adaptive information systems enable us to reason with and influence changes to objects and schema, based upon an explicit declaration of the relation between observable events and actors on one hand, and the changes on the other. These relationships are termed legal relationships. Legality is introduced as follows.

**Definition 3.18** An object-state of object \( id \) at time \( t \) is a conjunction of logical propositions stating \( a(id) = value(T) \) for every \( T \) in \( val_set(a)(att_val_set(id)) \) such that \( t \in interval(T) \).

**Notation:** \( A(O) \mid_{t_1} \) or \( Object\_State(O) \mid_{t_1} \) signifies that the (conjunction of) proposition(s) A is true of the object(s) O at time \( t_1 \). A may, in a specific case, specify the object state of the single object O.

Implicitly, an object-state specifies the membership of the object in one or more intensional collections at the given point in time. For all \( C \) such that \( Object\_State(O) \mid_{t_1} \Rightarrow Defn(C) \), we say \( (C \in Collections(O)) \mid_{t} \) or \( (O \in Member\_of(C)) \mid_{t} \).

**Definition 3.19** A snapshot at time \( t \) is a conjunction of object states for some collection of objects at time \( t \).

**Definition 3.20** An event denotes a set of co-incidental changes in observable attributes of a collection of objects.

**Definition 3.21** A snapshot A at time \( t_1 \) is said to be legally related to a distinct subsequent snapshot B at a time \( t_2 \) if:

1. \( t_1 \) is weakly antecedent to \( t_2 \): no \( t \in t_2 \) precedes any \( t' \in t_1 \)
2. If \( A(O) \mid_{t_1} \) is explicitly declared to necessarily imply \( B(P) \mid_{t_2} \), and each object in \( P \) has a tight or loose binding with some object(s) in \( O \).
3. There exists an event \( E \) occurring at time \( T' \) such that both \( [t_1, T'] \) and \( [T', t_2] \) are bonafide intervals.

This definition implies that the subsequent state in a legal temporal relationship follows the antecedent state, that legal relationships are explicitly declared, not implied by definitions, that antecedent and subsequent states are separated by observable change, and that
the objects are bound by necessity not mere coincidence. These are inspired by causality (chapter 1) under observability due to the finiteness of the modeled domain. D'alphabet combines legality, time intervals and membership to define the notion of reaction.

Definition 3.22 A reaction \( R \) is an event at time \( T \) which establishes a legal temporal relationship between an object \( O \) in a class \( A \) and an object \( P \) in a class \( B \), if the following are satisfied:

1. \( O \in \text{Members-of}(A) \) immediately prior to \( T \).
2. \(! (P \in \text{Members-of}(B)) \) immediately prior to \( T \).
3. \( P \in \text{Members-of}(B) \) at \( T \) and immediately following \( T \).
4. The object-state implied by (3) above is legally related to the object-state implied by (1) above.

Reactions are the foundational notion of dynamic relationship between object states based on legal change used to reason with dynamic domains (objects) and representations (schema). Though the definition refers to objects as members of one class, it can extend without loss of generality to extensional collections, each of which may be a member of more than one class, both before and after the reaction. Reactions are specialized in the following chapters in order to specify the syntax and semantics of dynamic relationships between objects and schema, both in informational and computational terms.

Notation: \( S = \text{Source-Class}(R) \) refers to the legally antecedent class in the context of a reaction \( R \).

\( T = \text{Target-Class}(R) \) refers to the legally subsequent class w.r.t. \( R \), which is denoted. \( S >> T \).

3.6 Chapter Summary

Chapter 3 is a prologue, where we refer to existing definitions from previous research to establish the basic terminology used throughout the thesis, and establish the framework for the main contributions. Objects, definitional and factual attributes, and interval relationships are derived from previous work in temporal and object-oriented databases.

In terms of contributions, this chapter proposes the framework itself, basic dynamic modeling concepts, namely object-states, legal dynamic relationships and reactions, and the concepts used to address change in relation to user's needs, namely utility and bindings. This framework is instrumental in separating computational devices common to objects and schema from modeling devices specific to either, and further, to separate modeling devices common to all domains from extensible domain-specific devices. The remaining contributions underlie the model of change presented in chapters 4 through 6.
Chapter 4  D'DIALECT(O) - Object Dynamics Language

The framework introduced in chapter 3 enables us to separate object dynamics from schema change, and domain-specific from domain-independent aspects. We start with the easier problem of representing object dynamics. In terms of modeling, the primitives proposed in this chapter represent our major contribution, though they are extended to schema in chapter 8. In this chapter, we assume that the observer's conceptualization of a domain in terms of the classes, and their intensional definitions, does not change. These classes are used to delimit the domains over which queries and updates are applicable. As observable attributes change, certain objects cease to be useful as members of their prevailing class(es) (They may be more useful when classified in a different collection.). Such changes are common in planning and design domains, and constitute an important source of dynamic integrity constraints.

Unless the information management system is aware of the legal, temporal and classificatory constraints of the domain, it is unable to maintain overall consistency under incremental attribute update. Neither is it able to answer questions regarding necessities and possibilities, or infer along temporal lines. D'dialect(O) is used to explicate these aspects of the domain. It consists of a dynamic definition language (sections 4.1 through 4.7) and a dynamic query language (section 6.7).

The dynamic definition discipline outlined in this chapter and the next helps the user to define the static and dynamic behavior of the domain. Accordingly, it includes:

- Class definition language to define the relevant grouping of objects in the domain.
- Reaction definition language to define legal dynamic behavior in the domain.
- Some guidelines for data modeling combining both class and reaction definitions, which helps to define schemas that account for the expected dynamic behavior.

4.1  Classifications Definition Language

The classifications definition language is a high-level language for defining intensional collections which is oriented towards describing change. D'alphabet included the definitions
CHAPTER 4. D'DIALECT(0) - OBJECT DYNAMICS LANGUAGE

of the static building blocks called object, identity, definitional and factual attributes, inten-
sional and extensional collections, tight and loose binding, and a notion of utility of an
object. Collections and bindings have some common interpretations in the static description
of objects, which comprise the essential static definitions of d'dialect(0). Attribute types and
notions of utility vary widely depending on the types of definitions required. Accordingly,
specializations of these notions comprise the static modeling features of d'dialect(0).

4.2 Essential Static Devices

The two most common methods of organizing an observer's view of a domain are
founded upon classes and types. Type often refers to the intensional definition of preferred
object groupings in the universe. Class refers to the extensional grouping of objects implied
by the definitions. Subset relationships among the clusters may be implied by the type defi-
nitions as well. We treat class and type based descriptions as parallel ways to conceptualize
a domain. (Devices for defining schema are addressed in chapter 8).

A single device is therefore provided for the static categorization of objects - the object
class. Any entity can be seen as an object class (Corella [50] uses the term 'intension' for a
similar concept), though each individual user may prefer to see some entities as 'types', and
others as 'instances' of the defined types. The notion of class of objects as defined below is a
weak concept to start with, as it extends the notion of extensional sets of objects modestly to
allow set-based reasoning with objects under change. The definition of a class does not
include the ability to relate the values of more than one attribute in a single proposition.

Definition 4.1 An object class C is a collection of objects specified by a set of
definitional attributes Defn_Arrt(C), and dynamic integrity constraints
Condition(C, X) associated with each X ∈ Defn_Arrt(C). C is defined by a
well formed formula Defn(C) in disjunctive normal form, which is a
conjunction of Condition(C, X) for each X ∈ Defn_Arrt(C).

Notation: Class(Id)_t or Class(ID, t) denotes the set of classes that an object Id belongs to at time t.
Class_Defn(C) refers to the first order logic formula for the class definition. It is synon-
ymous with Defn(C).

Definition 4.2 An object O is said to belong to C during time interval T if :
1. A = Att_Val_Set(O)
2. V = ∪ Val_Set(a)(A) ∀a ∈ Defn_Arrt(C)
3. V ⊆ V is the set of value intervals v ∈ V, such that
Overlaps(Interval(v), T) is true.

4. C' is the conjunction of logical assertions of the form a(O) = Value(v) \forall a \in \text{Defn}\_\text{Attr}(C) and \forall v \in V'.

5. C' \vdash \text{Defn}(C).

O is then said to be an instance (or member) of C during T.

**Notation:** Class\_Mem(Id, C) denotes the sequence of maximal intervals during which object Id is a member of class C. Another notation for the above assertion is that Object\_State(O) \vdash \text{Defn}(C).

The definition above expresses the condition that if the conjunction of all the assertions regarding values of definitional attributes that overlap time T (i.e. a(O) = v \mid T) implies Defn(C), then O is a member of C at T. This definition is usable in reasoning with objects' memberships in classes in the data model proposed. The idea is expressible simply as follows. An object O belongs to a class C during an interval T if the definition of C is satisfied by the values of the attributes of O which are current during T.

**Example 4.1** A line card in a telecommunications maintenance domain may be in one of two classes (or object-states) OOS (out-of-service) or BSY (busy) defined by the *definition attribute* Service\_State as follows:

- **OOS:** Service\_State = Faulty (dynamic integrity constraint)
- **BSY:** Service\_State = In\_Use (dynamic integrity constraint)

If Service\_State remains In\_Use from [1, 2) and then becomes faulty from [2, t_n), then the following definitions apply:

1. \( A = \langle O, \{ \text{Service\_State}, \langle \text{Faulty}, [1, 2) \rangle, \langle \text{In\_Use}, [2, t_n) \rangle \} \rangle \)
   
   > is the attribute value set of O.

2. \( V = \langle O, \text{Service\_State}, \langle \text{Faulty}, [1, 2) \rangle, \langle \text{In\_Use}, [2, t_n) \rangle \} \)
   
   > is the value set of A.

3. \( V' = \{ \langle \text{Faulty}, [1, 2) \rangle \} \) is the set of temporal value intervals that overlap \( T = 1.5 \) and are definitional attributes of O.

4. \( C' = \{ \text{Service\_State}(O) = \text{Faulty} \} \) is the set of assertions associated with \( V' \)

5. \( C' \) satisfies the definition of class OOS at \( T = 1.5 \); thus O is a member of OOS at \( T = 1.5 \).

### 4.2.1 Bindings of Objects to Attributes and Classes

In the context of objects and their classification, the concepts of tight and loose binding are interpreted in three different ways:
1. The binding of an object O to the members of Collections(O) is a bidirectional loose binding - neither is existence dependent on the other.

2. The relationship between an object-state and its essential aggregate subparts, including definitional attributes, is a tight binding while that between an object and its factual attributes is a loose binding.

3. The existence of certain object instances (and classes) is existence dependent on other instances and classes, in order to preserve certain desirable integrity constraints and abstract views of the information. When certain objects or abstractions become irrelevant, others which are tightly bound to them lose their reason for being as well. This manifestation of binding is discussed in Proposition 4.1.

The state of an object’s membership in a specific class is closely related to the definitional attributes and essential aggregate subparts of the class. Since an object-state is implicit (i.e. it does not have an identity) in an object’s attribute values, we use the special meta-attribute utility of the object as a function of the utility of its definitional subparts. Implicitly, the state of an object O’s membership in a class C has a high utility at time T, if O obey’s the definition of C at T. Utility may or may not be bound to the object’s identity (Proposition 4.2), i.e O’s membership in C at T is not related to it’s existence as a recognizable object. The tight binding between the existence (identities) of two distinct objects is modeled through identity evaluator (Skolem) functions, so that the dependent object’s identity is undefined if the related object ceases to exist In dynamic domains, this concept is naturally associated with the process of creation and destruction of objects in reactions.

**Proposition 4.1** Identity id(O) is interpreted by a uniquely quantified (3!) Skolem function of the form f(i_1, ..., i_n) where i_j, j = 1 to n are the identities of object-states that are tightly bound to O. If i_1, ..., i_n are the identities denoting objects which initiate a reaction where objects are created, then the expression above may denote the identities of the objects produced as a result of the reaction if they are tightly bound to the objects involved as antecedents to the reaction event.

1. If being a Skolem function, each f(i_1, ..., i_n) denotes a distinct identity.

2. In dynamic domains where objects are created and destroyed, f can be looked upon as an identity generator which generates a new identity (object) f(i_1, ..., i_n) from the component identities (objects) i_1, ..., i_n.

3. f can then be one of two types. A **constructor** function does not destroy the input identities i_1, ..., i_n, while an **evaluator** function creates a new identity and destroys access to the input identities.
Proposition 4.2 If the existence of an object itself is linked to the satisfaction of certain constraints on its essential aggregate subparts, then \( \text{id}(O) = f(i_1, ..., i_n) \) * \( u(O) \) where \( u(O) \) denotes the utility of \( O \) (section 4.3.1).

Proposition 4.3 If a class \( C \) is intensionally defined, then \( C \in \text{Members of}(O) \) if \( \text{Object State}(O) \) \( \vdash \) \( \text{Defn}(C) \). If it is not intensionally defined then the relationship of equivalence does not hold.

The implications of the above propositions are relevant to object dynamics (section 4.5).

Example 4.2 (1) If \( E \) is the identity of an engine, then \( C = f(E) \) may be the identity of a car, such that \( f \) is a constructor. Hence \( f^{-1}(C) = E \). Thus a car and its engine are loosely bound, and the engine may survive the car’s destruction.

(2) If \( E \) is the identity of an egg, then \( O = \text{Rnd!(E)} \) may be the identity of its omelette, where \( \text{Rnd!} \) (unique random number function) is an evaluator function. Therefore, by definition \( \text{Rnd!}^{-1}(X) = \bot \). Hence an egg and its omelette are tightly bound.

4.3 Attribute Semantics

In the discussion above, we abstract attributes through functions, making no distinctions among different kinds of attributes. In practice, the semantics of static descriptions are carried by the distinctions associated with different kinds of relationships. Common examples include generalization and aggregation relationships among classes, and instance relationships among classes and instances. Researchers have pointed out that many common static relationships have been interpreted differently across domains, so that a uniformly well understood semantics cannot be assumed. When static and dynamic modeling are combined, some semantic aspects of attributes that are dynamically determined can be specified in the static schema. We model these aspects as static features incorporated into static models. Example 4.3 suggests the importance of associating strict semantic implications with all relationships. In the case of domains with dynamic behaviors, the semantics of the static features are captured by specializing attribute functions as follows:

- differentiating among some basic attribute function types to reflect primitive distinctions among the semantic implications of different dynamic relationships.
- providing the computational semantics of the relationship in terms of ATNs (chapter 6), since the semantics inherently depend on behavior under change.

associating a name with the encapsulated definition.
Example 4.3  Static aggregation relationships between two object instances may carry containment semantics for attributes such as area. Alternatively, aggregation may carry attribute composition connotations (see (a) below). Similarly, generalization relationships among object classes may or may not imply disjointedness of sub-domains (as in (b) below).

4.3.1  Dynamics-based classification of attribute functions

The vehicle for carrying information about relationships between objects in our framework is the attribute, represented by attribute functions. In section 3.4.1, we distinguished definitional attributes from factual attributes, and claimed the distinction to be basic to any static description. In order to facilitate distinctions among static relationships, we recognize some further intuitive distinctions among attribute function types which can be used to differentiate their behavior under change (this research is reported in [74][67]).

Notation:  Dormant_Attr(C), Expiring_Attr(C), Eval_Attr(C), Constr_Attr(C) refer to the sets of attributes of the different distinguished types related to class C.

FIGURE 3.4  Subclassification of Attribute Functions
The distinctions introduced in this section are seen as static features, since they may not be relevant to every attribute. Figure 3.4 captures the basic set of static features. The classifications in the figure are orthogonal with respect to the distinction between definitional and factual attributes, and each other. They are also the basic device for the integration of static and dynamic considerations in a model.

4.3.1.1 Dormant and Expiring Attributes

Dynamic domains add the possibility of an object being an instance of a specific class at two different intervals of time, in addition to the static distinction between two different object-states within the same class at the same time. An Expiring type attribute captures those aspects of an object's definition or informational content which are tied to a particular instance of its membership in a particular class. If an expiring attribute \( a \) is defined for an object \( o \) during the time that \( o \) is a member of a class \( c \) at time \( t \), this does not constitute a necessary or sufficient condition for any similar fact, such as attribute \( a' \) being defined on \( o' \) while \( o' \) is a member of class \( c' \) at time \( t' \).

An expiring attribute differs from a Dormant attribute \( a \) when \( o \)'s membership in a class \( c' \) at time \( t' \) is related in a necessary or sufficient fashion to its membership in class \( c \) at time \( t \). In more formal terms, a dormant attribute captures the situations where \( a(o) \) at \( <c,t> \) is dependent on \( a(o) \) at \( <c',t'> \).

The distinction between dormant and expiring attributes is significant and commonly interpreted in dynamic descriptions as follows:

If \( o \) is a member of class \( c \) during two different intervals \( t_1 \) and \( t_2 \). If \( a \) is a Dormant attribute associated with \( c \), then \( a \) is made current during \( t_2 \) with the value that it had during \( t_1 \). If, however, \( a \) is an Expiring attribute, then \( a(o) \) at \( t_2 \) may not be predicated upon \( a(o) \) at \( t_1 \) - a new value is made current.

Though this static distinction is primarily motivated by dynamics, it is similar to certain static relationships insofar as its realization in terms of a uniform computational structure is concerned. An analogue of dormant attributes in static modeling is environmental variables in aggregation hierarchies (for example, the fuel capacity of a car is the fuel capacity of its component part - its fuel tank. However, fuel capacity may be undefined for the drive system, an intermediate aggregate class between 'fuel tank' and 'car', in the aggregation hierarchy describing a car). In terms of the general graph structure illustrated in figure 3.1, both these computations are interpreted similarly as attributes defined on paths in the graph.
Example 4.4 An Expiring attribute is *SIN_of_temporary_residents*; it expires every time a person ceases to be a temporary resident, and is re-initialized. A Dormant type attribute is exemplified by the Medical Services Plan (MSP) of British Columbia ID_no_for_self-financing_beneficiaries. Persons who self-finance their medical coverage during disjoint intervals of time have the same BC MSP number during those intervals. The number differs during the interim intervals, when they may participate in BC MSP by other means (such as through salary deductions at source).

Attributes are expiring by default.

### 4.3.1.2 Evaluated and Constructed Attributes

The second orthogonal distinction among attribute types distinguishes *evaluated* attributes from *constructed* ones. Attributes, which have signatures of the form \( a(o) \rightarrow v \), are defined by functions of the form \( a(a_1, \ldots, a_m) \). If \( a_1, \ldots, a_m \) are attributes of different objects, the function \( a \) evaluates the attribute value of \( a \) based on those of \( a_1, \ldots, a_m \). \( A \) can be a *constructor* or an *evaluator*.

*Constructor* attributes specify necessary and/or sufficient constraints among attributes. For example, the constructed attribute constraint \( a = b + c \) for the attribute 'a' specifies a constraint which must be dynamically maintained over time, as \( a, b \) or \( c \) change. *Evaluator* attributes embody static constraints, which are only enforced during initialization. If the attribute 'a' were specified to be an evaluator attribute with the constraint "\( a = b + c \)", then \( a \) is constrained to be initialized with the value "\( b + c \)", but may subsequently be modified.

Attributes are evaluated by default.

### 4.3.1.3 Utility

The essence of dynamic systems is that as the properties associated with objects change, so should their classifications. In our framework, the stability of an object in its current form is captured through the notion of *utility*. Utility is a distinguished attribute discussed in section 3.4.1. The utility of every object and representation mechanism is defined in terms of the utility of its component parts. As the component parts undergo change, so does the utility of the aggregated object. The measure of utility can be used as a basis for enforcing changes in object classifications, and in the schema.

*Proposition 4.4* Utility\((O, \text{Class}, \text{Time})\) is a distinguished definitional attribute \( u \) associated with every object \( O \) in every class in which \( O \) belongs, at all
times. It is a constructed and dormant attribute.

Utility \( u(o) = f(u(o_1), u(o_2), \ldots, u(o_n)) \)

where \( a_i(o) = o_i \) for distinct attributes \( a_i(o), i=1 \) to \( n \).

The schema in terms of which objects are recognized should itself adapt to different user preferences, and this is the situation where the concept of utility is important (Appendix B and chapter 8 discuss this issue). However, we extend the notion to object classification for completeness.

In the case of object migration where we assume the existence of a well-defined ontology of classes and relationships, the utility function \( f \) is defined trivially. Since intensional class definitions specify a deterministic criterion for membership and therefore change in membership, the utility of an object as a member of a class is maximum exactly when it satisfies the class definition and minimum when it does not. Hence, the utility function for objects, \( f \), has the following properties:

- Range of \( u \) is \( \{0, 1\} \)
- \( u(O_i) = 1 \) if \( (\lambda a_j)(\text{Condition}(C, a_j))(O_i) \implies 1 - T_i = 0 \) otherwise. I.e. 1 if the condition on attribute \( a_i \) is satisfied when \( a_i \) is substituted by \( O_i \) in the definition of \( C \), else 0.
- \( f(u(o_1), u(o_2), \ldots, u(o_n)) = u(o_1) * u(o_2) * \ldots * u(o_n) \), or a simple multiplication of the component utility values.

In effect, \( \text{Utility}(O, \text{Class}, t) = 1 \) if \( \text{Object-state}(O) \implies 1 - \text{Defn(Class)} \) and = 0 otherwise. In the case of schema, the utility function is more granular, in support of the greater range of flexibility demanded by representations, as explained in section 8.6.1.

**4.4 Reactions Definition Language**

In section 4.2, we introduced the essential static devices. This section introduces the essential dynamic primitives for modeling change, which is the main contribution of this thesis. We propose to integrate static and dynamic modeling as follows. First, we give first class status to those aspects of the domain which determine its dynamic nature among the static modeling constructs (and vice versa). Next, we propose an integrated modeling framework. Finally, we use a common formalism (d'foundation) for the computational interpretation of both static and dynamic modeling primitives. Thus, dormant and expiring attributes, which are motivated by dynamics, are introduced as static modeling constructs. Similarly, certain convenient static devices such as inherited attributes can be interpreted in
terms of dormant attributes. In chapter 5, we introduce an integrated static/dynamic modeling formalism, and chapter 6 introduces d'foundation.

Change in the world is comprised, at least in part, of change in the entities that make up the world. We distinguish two fundamental ways in which entities in the world change. One way is that these entities appear and disappear. Such change is modeled, at the lowest level, by the insertion into and deletion from the model of the objects which represent the entities. However, low level insertions and deletions do not represent the legal and temporal relationships that often hold between the disappearance of one entity and the appearance of another, or between the existence of one entity and the appearance of another.

The second way in which an entity may change is that its properties may differ from one time to another. In other words the entity exists prior to the change in question, and continues to exist subsequent to the change in a somewhat different fashion. Such change is modeled, at the lowest level, by the updating of the attributes of the objects which represent the changing properties of the entities, if such attributes exist. At a higher level, this second type of change may require the reclassification of the object in question, if the property which changes is one which the user community employs to distinguish between different classes.

We identify two key factors which seem to characterize different kinds of change which need to be modeled in information systems: preservation and identity.

4.4.1 Preservation and Identity

Different kinds of changes in a domain of discourse are differentiable on the basis, firstly, of whether the change preserves the constraints and conditions which prevailed before the change took place. Orthogonally, changes are distinguishable on the basis of whether the objects recognizable after the change is affected are identical to those identifiable before the change.

The changes which are discussed in the available literature invariably, though unconsciously, respect these distinctions. Since the semantics involved in these distinctions, as well as the devices required to model and reason with changes of these different kinds are differ-
ent, we distinguish them explicitly in our proposed formalism. Figure 3.5 shows our classification of the essential changes in a domain, categorized along the two axes mentioned.

<table>
<thead>
<tr>
<th>Identity Preservn.</th>
<th>Changing</th>
<th>Non-Changing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preserving</td>
<td>PRODUCTION</td>
<td>EXTENTION</td>
</tr>
<tr>
<td>Non-Preserving</td>
<td>TRANSFORMATION</td>
<td>EVOLUTION</td>
</tr>
</tbody>
</table>

**FIGURE 3.5** Essential Dynamic Definitions

The concept of *reaction* given in section 3.5 is specialized by these four *essential dynamic modeling devices* (production, transformation, extension and evolution) which constitute the core of our reaction definition language. Apart from capturing two of the distinctions which are intuitively significant in the conceptualization of dynamic behaviors, these abstractions differ in their realization through d'foundation.

The reaction definition language is a high-level language to describe the dynamic behavior of a domain. A key requirement, therefore, is that it should be simple to use. Accordingly, it consists of just the four primitives named in section 4.5, but it is designed to be extensible in conjunction with the classifications definition language through its computational interpretation using ATNs.

### 4.5 Essential Dynamic Devices

As in the case of static definitions, we recognize that certain aspects of change of classification of domain objects are essential in describing dynamic behavior. As per our objectives outlined in section 3.1, we distinguish these as *Essential Dynamic Devices*. Some other aspects are less ubiquitous but of critical value in capturing the semantics of change in certain specific domains. These are collectively called *Dynamic Features*. The four primitives outlined above constitute the essential dynamic definitions.

**Evolution** occurs when an object ceases to be a member of a class $A$, and becomes a member of a different class $B$. 
Extension occurs when an object becomes a member of a different class B, but continues to be a member of its earlier class A.

Production occurs when an object o in class A creates a distinct and distinguishable object o' in class B, and both o and o' continue to exist after the reaction.

Transformation occurs when an object o in class A expires to create a distinct and distinguishable object o' in class B.

Notation: In terms of relationships, the primitives above relate objects and object-states at different intervals in time. Source(R, O-S) is the terminology used to refer to the entities related by the reaction R which are legally antecedent to the object-state O-S. GSource(O) is the term used to refer to the source of the generation reaction which led to the creation of the object O. Self is a special identity term which refers to the current object-state. Analogously, Target(R, O-S) and GTarget(O) refer to the legally subsequent objects or object states of a reaction.

4.5.1 Transition

For reasons of historical consistency and convenience, we group the first two reaction types, where no objects are created or destroyed, under the subcategory transition, which the latter two reaction types which involve creation and/or destruction of objects are classified as generation reactions. (The transition and generation sub-categories are intuitive aids, not directly used for modeling.)

More formally, then:

Definition 4.3 A transition is a reaction established at time point T, which relates Object-state(O) I T to Object-State(P)I t where t immediately follows T, such that O ∈ Members-of(A)I T and P ∈ Members-of(B)I t and id(O) = id(P) (O and P are the identical object).

In other words, a transition models change in the classification of objects as a result of events. Thus, in terms of a dynamic schema construct, transition links target object classes (B) to source object classes (A) with respect to a particular object by modeling change of instances of the source class into an instance of the target class. It represents the becomes-a

\footnote{See [67],[68],[69],[74] for reports of this work.}
relationship between the object-state which is changed and the object-state which results from the change.

In practical dynamic modeling, an additional level of specialization of the transition concept is useful, which relates to the philosophical distinction between 'class' and 'role' in the observation of common experience [11]. (This distinction is perhaps the driving force behind the more constrained notions of 'abstract' and 'concrete' classes in static descriptions). The basic idea behind this distinction is that an object may be a member of more than one class, some of which are 'dependent' on others. Moreover, an objects membership in some of these classes could change over time, and may or may not affect its membership in other classes.

This distinction is made through the two categories of transition, viz. evolution and abstraction, which are based on whether or not the object undergoing the transition preserves its status as an instance of the source class.

Definition 4.4 An evolution is a transition at time T, relating the existence of an object O as an instance of source class A immediately prior to T, and its existence as an instance of the target class B at and immediately following T, with the following additional condition:

\[ \neg (O \in \text{Members of}(A) \cap T) \lor \exists t \ I \_\text{Foll}(t, T), O \in \text{Members of}(A) \_\{t\} \_\cup \]

i.e. O is not a member of its source class with respect to the transition in any interval immediately following the transition.

An evolution is thus said to occur when the transiting object ceases to be an instance of the source class as a result of the transition. Evolutions may be used to model transitions among the roles of a class, as well as significant changes involving the primary classes which an object may belong.

For example, when an object representing an applicant changes to reflect the acceptance of the applicant into a graduate program, it undergoes an evolution; that is, it ceases to be an instance of the applicant subclass and becomes an instance of the student subclass (alternately, role). We represent an evolution in conceptual diagrams by a heavy arrow from the source (class) node to the target (class) node, (see Figure 3.6). (In running text, an evolution is represented thusly: ==>.)

An extension, in contrast to an evolution, is a transition which generalizes the notion of role-formation. When an object enters into a role associated with a class, by definition, it remains a member of the class. An extension models the situation of an object becoming a member of a class due to its membership in some other class, while retaining its existing
class memberships. Since in our formalism, we do not differentiate between class and a role attached to a class, extension defines a more general relationship between classificatory groupings. Formally:

**Definition 4.5** An extension is a transition at time T, relating the existence of an object O as an instance of source class A immediately prior to T, and its existence as an instance of the target class B at and immediately following T, with the following additional condition:

\[ O \in \text{Members}_\text{of}(A)_{T} \land (\exists t \ I_{\text{Foll}}(t, T), O \in \text{Members}_\text{of}(A)_{t}) \]

For example, when an alumnus with a Master's degree applies to the Ph.D. program, the transition of the object representing the alumnus into an instance of the applicant sub-class is an extension, because it is still of interest to include the object among the class of alumni of the master's program. An extension is represented in conceptual diagrams as a heavy arrow with a slash through the shaft, (in running text: \(=\Rightarrow\)).

An transitionary action which changes an antecedent state to a consequent state is a required action if the antecedent or consequent states are necessary components of the source or target class definitions. If a required action is omitted during a transition from class A to a class B, then the object does not fulfill the conditions associated with the extension or evolution as dictated by the definitional attributes of A and B.

Thus, required actions involve operations on definitional attributes, and may be automatically generated from the high-level transition specification, at least in part. For example, definitional attributes of class B can be automatically attached to an object transiting to that class; however, it may or may not be possible to automatically instantiate such an attribute.

An action associated with a transition is an optional action if its omission does not prevent the transiting object O from fulfilling the conditions associated with the definitional attributes of the source and target classes A and B to prevent the transition. Optional actions may operate on factual attributes or attributes unrelated to the transiting object.

### 4.5.2 Generation

Transition abstracts a dynamic relation between objects that share the same identity. It is useful in modeling, for example, students' transition through a graduate program. In such domains, the identity of a student object is maintained through all changes. No person object is either produced or consumed in a transition of state. As a corollary, all transitions are one-to-
one. For example, one applicant object is accepted, causing the same object to become one student object.

These assumptions are violated in a very large class of interesting problems which involve object migrations, exemplified by the production planning problems. These call for a corresponding non-identity relation, which we call generation. Characteristics of production planning-like domains include:

- *New objects are routinely created and destroyed.* Mill orders have their own identities apart from the sales order(s) that created them, for example. They are not the same object as the sales orders.

- *The input items of a process may or may not survive the process.* A sales order lives on even after the mill orders have been created. It may undergo subsequent modifications which affect the mill orders it spawned. However, the oleum and water which combine in certain proportions to produce sulfuric acid cease to exist after the process.

- *Items are created and destroyed in all cardinalities.* In general, any M items may be combined in a process to create N other items.

- *Some processes are (pragmatically) reversible, while others are not.* If a mill order can not be completed at one mill due to machine breakdown, the conversion of the original sales order to the mill order can be reversed, the current mill order destroyed, and a new mill order created to be executed by a different mill. However, once sulfuric acid is produced, the process can not be reversed to yield oleum (at a given plant).

- *Attributes of source and target object classes are interdependent.* The tonnage of sulfuric acid produced depends on the tonnages of raw materials input into the process. Conversely, the date of execution of a mill order determines the date of delivery of the sales order.

The above characteristics are enumerated in their relationship to production planning processes. Of these, the first two characteristics are essential distinctions which may be made in any reaction situation where creation and deletion of objects is involved. Consequently, we classify them under the category of essential dynamic definitions and capture the distinctions through the dynamic modeling device generation. The other characteristics are arguably domain-dependent, and we therefore categorize them as dynamic features, which are realized using auxiliary modeling facilities of d'foundation. Being thus realized, the set of features may be easily customized through the definition of macro keywords with associated d'foundation realizations.
Generation models the dynamic relationship of creation of objects in (target) classes from other objects in (source) classes where there is no shared identity between the source and target instances. It represents the yields-a relationship between the source and the target instances, and is useful for modeling the legal and temporal relationships that are involved in the appearance and disappearance of entities in the world.

Definition 4.6 A generation is a reaction established at time point $T$, which relates object-0 to object $P$ where $t$ immediately follows $T$, such that $O \in \text{Members-of}(A)_{t}$ and $P \in \text{Members-of}(B)_{t}$ and $\text{id}(O) \neq \text{id}(P)$ (O and P are distinct objects).

Generation, as defined here, relates a single object in a source class to another single object in the target class. However, M:N relationships are easily represented through the use of the M:N modeling feature mentioned in section 4.6.1.

As in the case of transition, two types of generation are distinguished based on the survival of the object(s) input into the generation process. In the case of generation, the issue of survival of the input objects is much more evidently important, since this device is used to model the physical (and sometimes irreversible) creation and destruction of objects in the real world.

Definition 4.7 A transformation is a generation at time $T$, relating the existence of an object $O$ as an instance of source class A immediately prior to $T$, and the existence of a distinct object $P$ as an instance of the target class B at and immediately following $T$, with the following additional condition:

$$\text{id}(O)_{t} = \bot \land \text{id}(O)_{t} = \bot, \text{Foll}(t,T).$$

$O$ does not exist and therefore has no attributes at $T$ and in any interval following $T$, and is therefore also not a member of A at $T$ or in any interval immediately following $T$.

A transformation occurs when the instance of the source object class ceases to exist while creating an instance of the target class. For example, when an object representing a unit of oleum enters a Gay-Lussac tower, to implement the production of sulfuric acid, it undergoes a transformation; that is, it ceases to exist as a member of the oleum subclass, as does the unit of water which also goes into the production of sulfuric acid. As per most conceptualizations, the unit of sulfuric acid with its own batch number which comes into existence is an entirely distinct object from the oleum or water that went into its creation. (The identity of the new object is perhaps derived from that of the oleum, perhaps not.) In running text, a transformation is represented by: -->.
A production denotes the generative correspondent of the extension dynamic relationship.

**Definition 4.8** A production is a generation at time $T$, relating the existence of an object $O$ as an instance of source class $A$ immediately prior to $T$, and the existence of a distinct object $P$ as an instance of the target class $B$ at and immediately following $T$, with the following additional condition:

$$O \in \text{Members}_\text{of}(A) \cap T$$

A production is said to take place when the source object survives the generation process.

Each reaction may carry the qualification *during*, signifying that the interval in which the target object-state exists replaces the validity interval of the source object-state. We illustrate through an example where a sales order generates corresponding mill orders in a typical paper production process. The figure shows how a sales order may be suspended periodically until it is collapsed.

![Diagram of Essential Dynamic Definitions](image)

**FIGURE 3.6** Essential Dynamic Definitions

The sales order ceases to be active in reactions only if it is collapsed (summarizing all sales order activity into one record) later in the lifecycle of order processing. In the interim, a sales order and its generated mill orders may undergo their independent reactions and migrate through different classes nominally independent of each other. Note however that the identity of a 'production' or 'transformation' product can be existence-dependent on its source object, as explained in proposition 4.1, as may be its attribute values. In text, a production is represented by: $\rightarrow$, and in diagrams, by a dashed arrow with a slash across it. The figure above illustrates a simplified flow of records in a production management system. A sales order may be suspended and reactivated any number of times, undergoing evolutions *during the lifetime of the original sales order*. At some point, it may produce a material order.
without being destroyed itself. At yet another date, it may be collapsed as an archival entity summarizing all its activity, when it is no longer actively referred to. The material order may be transformed into a Goods Requisition order that authorizes the withdrawal of material from the warehouse. Since the material order is not needed once the goods are withdrawn, it need not survive this reaction. Once the goods are withdrawn from the warehouse, the goods requisition may double as an invoice for the material, thus undergoing an extension.

4.6 Dynamic Features

The reaction definition language (RDL) ’s essential dynamic devices given in section 4.5 are instrumental in triggering changes to a database, and in making inferences regarding changes in the real world as modeled in an information base. The limited language represented by these essential definitions, however, falls short of intuitive adequacy in most practical cases. Many crucial distinctions in dynamic domain definitions are specific to particular domains and not universally applicable. Yet, in the domains where they are applicable, the usability of the dynamic modeling framework hinges crucially on the ability to model their specific semantics. We provide the ability to model these crucial aspects by making RDL extensible through adjectives. Adjectives qualify reactions to carry special semantics, and they are realized through the auxiliary modeling features in d’foundation.

For example, in many situations, it is of critical importance to explicate the difference between reversible and irreversible reactions. Similarly, in other situations, spontaneous reactions must be distinguished from those which need to be explicitly triggered by a user. Such aspects of dynamic relationships, however, are too numerous, potentially too detailed, and entirely absent from certain domains (for example, there are no reversible reactions in financial domains and no spontaneous reactions in telephony resource provisioning). Consequently, they can not constitute essential parts of dynamic modeling.

The potential set of adjectives is large. We illustrate the concepts through four useful adjectives which might be associated with dynamic definitions in production planning domains: reversibility, spontaneity and M:N reactions. In section 6.4, we illustrate their realization in terms of d’foundation.

Notation: Adjectives(R) refers to the set of adjectives associated with a reaction R.
The topic of specialized semantics for specific devices has been largely ignored even in the case of static modeling, where a small set of static relationships such as IS_A and aggregation are generally held to be sufficient. In fact, other researchers have recognized that different domains demand special semantics for variants of relationships like IS_A and aggregation. While the means described for dynamic definitions may be used in customizing static definitions as well, this is not the focus of this thesis.

4.6.1 Examples of Dynamic Features

4.6.1.1 Spontaneity

An important property of some real-world reactions, particularly in production planning scenarios, is spontaneity. Spontaneity refers to the tendency to effect a reaction as soon as a valid set of inputs is available, or to commit a potential input object to a future reaction as soon as it is created. If an application requires spontaneous generations, then the generation could be qualified by the constructed adjective Automatic. If spontaneity is to be modeled in an information system, the semantics of the automatic adjective need to reflect the characteristics rationalized as follows.

A spontaneous reaction will occur when the act of creation of an object-state immediately satisfies the pre-conditions of the outgoing spontaneous generation. A pre-condition for a reaction may be completed if it is equivalent to the definition of the source object class of the spontaneous reaction as in figure 3.7, causing a need for the affected object to leave the source class.

![Spontaneous Reactions Diagram](image)

**FIGURE 3.7** Spontaneous Reactions

If we consider that a precondition on a reaction is satisfied if the source class definition is violated, then a different way to state the above is as follows: The automatic keyword prompts the ATN (chapter 6) implementing d'foundation to modify the pre-conditions on outgoing spontaneous reactions to ensure semantic integrity.
Example 4.5  In a chemical process, a spontaneous reaction occurs when an unstable or volatile object (chemical compound) is produced. Similarly, even if an object (chemical) produced is stable by itself, its creation can further the possibility of a spontaneous reaction occurring in the context in which the object exists; i.e. the spontaneous reaction is triggered as soon as another ingredient is in place. An example of the first category is a chemical reaction which produces iodine at room temperature; immediately upon creation, the product vaporizes. An example of the second category is the release of ZnO (zinc oxide) on the surface of a V_{2}O_{5} (di-Vanadium Pentoxide) catalyst. Immediately as water is added to the reaction vessel, trace elements of sulphur in the water react with the ZnO to form ZnS. Though ZnO in the absence of water is stable, every unit of ZnO produced is immediately committed to the next unit of water available to react with it (also true for every unit of water). In terms of the above reasoning, the reaction is formulated as follows:

Definition w.r.t. outgoing reactions: \(~ (part1=ZnO \land part2 = \text{water})\)

In the example above, the dotted lines represent the action of aggregating units of ZnO and water as they are produced into the reaction input object. The definition of the object, however ensures that as soon as both objects (ZnO and water) are present, the automatic reaction producing ZnS must be triggered. It is also crucial to note that although many objects elsewhere in the domain would satisfy the definition ~ (part1=ZnO \land part2 = \text{water}), not every object would be considered a member of the class Input, since only those objects which enter class input after undergoing reaction R' are actually considered to be members of class Input. Non-spontaneous reactions may require the intervention of the user to proceed to completion. An example of this in the chemical domain would be the requirement of the user to manually raise the temperature of a reaction vessel before a certain reaction can take place. User intervention could be interpreted in terms of a boolean go ahead condition being manually set before the changes are effected. Non-spontaneous reactions may also proceed if an attribute update makes it impossible to maintain the source object-state.
4.6.1.2 *(Ir)reversible Reactions*

The second crucial distinction in practical systems is between reversible and irreversible reactions. Certain changes in the domain of discourse must be undoable, at least until some point in the future, whereas other changes can not be pragmatically undone. In some interpretations, an irreversible generation implies that the object(s) generated by the reaction can not lead back (informationally within the context of the computational mechanisms) to the object(s) which were involved in the generation process. Under this interpretation, there may be reactions which are neither reversible nor irreversible. Other interpretations hold reversibility and irreversibility to be mutually exclusive and exhaustive. We use the adjectives reversible and irreversible to qualify reactions as appropriate.

As in the case of spontaneous reactions, the semantics of (ir)reversibility is interpreted in terms of the modeling devices for static and dynamic descriptions in 'foundation. In contrast to the intuitions regarding spontaneous reactions, however, (ir)reversibility demands the ability to ensure that certain conditions are maintained after the reaction in question, rather than before. Thus, the translation of requirements to realize (ir)reversibility is made in terms of the actions and post-conditions associated with the primitives in 'foundation.

**Example 4.6** Reversible reactions in the process industry are exemplified by the production of paper from pulp. If the production of a particular unit of paper needs to be undone, a process called "repulping" may be applied. Repulping may be considered, from an information modeling viewpoint, to be a reversible reaction, if the historical record of the pulp -> paper generation is irrelevant after the paper is repulped - the pulp must be treated as if it had always been pulp. The same pulp -> paper process is irreversible if the paper later undergoes a 'glazing' reaction, because the chemicals added in the process of glazing damage pulp.

4.6.1.3 *The Essence of 'foundation Semantics*

The underlying semantics of the Augmented Transition Network (ATN) formalism on which 'foundation is based are that all the traversable links can be backtracked upon as in a parsing grammar. Under these assumptions, reversible reactions are modeled by regular ATN links. The keyword reversible causes all identity generation functions associated with the source identities to be constructors (which can be deconstructed) and ensures that all future functions involving the attributes of the source objects are bijections implying that they have inverses. These conditions are enforced through the postconditions associated with the reaction. Irreversible reactions imply that from the target object, the source object can not
be accessed or retrieved. In terms of the ATN, the actions associated with the reaction ensure that if the reaction is a generation, any identity creation function which defines the identity of the created object is an evaluator, not a constructor (i.e. can not be reversed). As well, the actions ensure that any functions applied to the source objects attributes in order to effect the reaction are one-way (irreversible) by applying a transformation to all such functions that have inverses. Figure 3.8 illustrates these ideas.

![Diagram of Reversible and Irreversible Reactions](image)

**FIGURE 3.8**

Reversible and Irreversible Reactions

Note that though irreversibility is enforced by one-way functions, the historical record of an object may nevertheless remain accessible for information purposes. The primitives only ensure that the reaction will not be reversible within the framework of high level dynamic specification defined by d’language. Note further that low level deletes and inserts can subvert the intent of reversibility and irreversibility (much in the same way as machine-level instructions can successfully undermine the constraints imposed by compiled HLL constructs for structured programming). The intention of this thesis is to provide a means to specify and enforce high-level dynamic constraints through a usable high-level language.

4.6.1.4  **M to N Reactions**

M:N reactions model situations where the co-presence of a number of objects of particular classes at a particular time and place legally lead to the co-presence of a different set of objects in different classes at the next relevant point in time (i.e. the restriction of single input and output objects are removed). In terms of a chemical analogy, co-presence of the correct measures of Zinc and Sulfuric Acid are required, at room temperature, before they can react to form Zinc Sulphate and Hydrogen. In this case, the inputs and outputs of the reaction are
entirely distinct entities.

A reaction is a single conceptual entity; all relevant information about it should be encapsulated within a single object as opposed to being dispersed among the descriptions of the objects involved in the reaction. This contrasts with schemes based on communicating finite automata as mentioned in section 2.4. We propose to model an M:N reaction as a 1:1 reaction $R$, with a complex source object $I$, aggregated from the objects which are input into the reaction, and a complex target object $O$, aggregated from the objects which emerge from the reaction. Both $I$ and $O$ are loosely aggregated from their parts, i.e. the identities of $I$ (resp. $O$) and its parts is a loose binding (definition 3.9). This means the subparts can survive even if $I$ (resp. $O$) expires. $R$ may be any kind of reaction, although most are transformations.

We use the adjective MtoN to qualify a reaction with multiple inputs and/or outputs. In the figure 3.9 below, this keyword prompts the system to aggregate the declared inputs and outputs of the reaction into the input and output objects $I$ and $O$ as shown.

![M to N Reaction Diagram](image)

The actual inputs and outputs (objects a through e) retain their individual identities distinct from $I$ and $O$. $I$ and $O$ serve primarily to relate a, b and c to d and e. For example, if the user would like to recognize the set of input objects and output objects of an M:N reaction apart from the objects themselves even after the reaction, then the preservation of the object $I$ can be ensured by declaring $R$ to be a production, and object-state $O$ could be allowed to exist indefinitely by defining no reactions sourced at $O$.

The attributes of the input and output objects, as well as the objects themselves can be referenced because $I$ (resp $O$) are aggregated from the input objects, which in turn are aggregated of their attributes. Hence the relationships between the attribute values of source and target object-states can be specified easily. It is more complicated to specify survival relationships among the input and output objects of an M:N reaction.

Some input objects in an M:N reaction may survive the reaction (perhaps in a different object-state) while others are consumed. For example, a catalyst in a chemical reaction sur-
vives the process, retaining identity and all its properties. Other reactants may lose their identity. In modeling this situation, we propose to use the adjective Survives_As(X) to qualify the arguments input into the M:N reaction. X is the name of one of the target classes of the M:N reaction. The figure 3.10 illustrates the situation described.

Three aspects worth noting are:

- Two target object-states which are members of the same class can be differentiated through the device of using different attribute names in the class O.
- I and O may be named or unnamed in the dynamic declaration, depending on the user's intent to refer to the collection explicitly or merely to use I and O as devices to group the source and target objects.
- If O is not significant, it can be 'killed' as soon as R is complete by specifying a spontaneous reaction with no target whose source is O, as in figure 3.11.

FIGURE 3.10  Persistence Across Destructive Changes

Though we have not defined the "null (kill) reaction" mentioned above as a dynamic feature, this can be done in a manner similar to the description of features above.

FIGURE 3.11  The "Null" or "Kill" Reaction

We propose the above dynamic features as useful devices for defining certain dynamic object bases. Yet, this set is necessarily incomplete in the sense that yet other intuitive relationships in evolving domains are surely neglected. Our research effort would make sense only if
we can show that these primitives can be interpreted in a flexible framework which is analyzable, extensible (restrictable), and capable of covering a useful expressive ground. Hence, we propose a formal, primitive framework for defining migration patterns and show how our primitives relate to this underlying basis. We study that basic framework with respect to its expressive power and complexity. In doing so, we define the limits of expressive power and the corresponding cost, of other intuitive modeling primitives which may be founded on the primitive framework.

4.7 The Structure of Formal D'Language Definitions

D'language provides a syntax to express the distinctions outlined in the preceding sections. In practice, we anticipate the need for a graphical tool to express the static and dynamic relationships in a domain. However, the essence of any such definition is expressible in a linear language such as the one below. Since the syntax is not the main emphasis of this thesis, we illustrate by examples at this point.

Example 4.7 An example fragment of a valid class definition, illustrating the use of the CDL constructs relating to attribute classifications and definitions follows:

```
Class Sales-Order
  Definition {
    PIN-Code: Integer,
    Dormant S-Ord-No: String,
    Dormant Constructor Due-Date = date(Source(*,Self)) - 5)
  }
```

This fragment is similar to class definitions in many object oriented languages. We highlight some differences:

- The keyword Definition distinguishes those parts of the declaration which are necessary and sufficient conditions for an object to be a sales order. In contrast, the status of all attributes in other languages is identical.
- The attribute Due-Date embodies an integrity constraint with respect to whatever entity is involved in an object reaching the state of being a sales order (the source of its self), presumably a letter of intent.
- The interpretation of Source and G-Source introduced later follows in chapter 6.
Due-Date is specified to be a Dormant attribute. This means that the first source remains valid throughout the lifetime of this sales order. This is better illustrated in the next example.

- S-Ord-No is Dormant. This implies that one logical sales order carries its number throughout its life, even if it becomes a member of some other class for some time in its history (see the following example)

- Since no integrity constraints are associated with PIN-Code or S-Ord-No, it is implied only that these attributes should be possessed by a valid Sales Order, without restricting the valid values they may have.

Example 4.8  
BC Medical Insurance number for non-immigrants on working visas is a value-dormant attribute. Every time such a person is on a work visa, his/her BC MSP code is the same. Between two working visas, the person may obtain BC MSP coverage by private contribution or other means. During these phases, the code could be a different random number.

Example 4.9  
In an telecom database at location A, the service profile of a subscriber could be assigned a value-dormant record-locator. If the subscriber moves through a series of locations (causing the personal profile to move according to "follow me" conventions), and then returns to A, the value of the record-locator would be restored. Thus, all pre-existing relationships with the subscriber record at A would resume validity, without requiring the record itself to be preserved. In the interim, the record could have been transformed and assumed other locators.

Example 4.10  
The example is extended by including reactions and adapted to a production planning domain below.

\[
\text{Class Sales-Order} \\
\quad \text{... Definition } \{ \quad \text{PIN-Code,} \\
\quad \quad \quad \text{Dormant S-Ord-No,} \\
\quad \quad \quad \text{Dormant Constructor Date=date(Source(*,Self)) -} \\
\quad \quad \quad \quad 5 \} \\
\quad \text{Factual } \{ \\
\quad \quad \quad \quad \ldots \\
\quad \} \\
\text{Class Susp-S-Ord} \\
\quad \text{... Definition } \{ \text{Susp-Order-No,} \\
\quad \quad \quad \text{Date = } \Phi \} \ldots
\]
Evolution Suspend DURING <Reversible>

... Input { Source(Sales-Order) } 
Output { Target(Susp-S-Order) } ...

Evolution Reactivate <Reversible>

... Input { Source(Susp-S-Order) } 
Output { Target(Sales-Order) } ...

- If O is created from a Letter of Intent dated 10 October, then O as a sales order is defined by the definitional attributes PIN-Code = XYZ, S-Ord-No = 1234, Due-Date = 5 October.
- If it subsequently becomes suspended during the interval of validity of the sales order. It is defined by the definitional attributes Susp-Order-No = 5678, Date = @.
- Thereafter, if it is reactivated, it gets back all its old attributes except PIN-Code, as per the logic below.
- PIN-Code has expired and a new one needs to be assigned.
- S-Order-No = 1234 because it was a dormant attribute.
- Due-Date = 5 October because it is a dormant attribute and it is 5 less than the originating Letter of Intent (not 5 less than the date on the Susp-S-Ord, which is the current originator).
- Since Due-Date is a constructor, if the original letter of intent has been redated to 15 October, the new due-date on the sales order becomes 10 October.

Example 4.11 The example below specifies the generation of both pulp and paper from two different types of wood chips in the presence of a catalyst which survives the reaction.

Transformation Paper-Make <Irreversible, Automatic, MtoN>

Input (Source(I-Class)) 
Output (Target(O-Class))

Class I-Class

... Definition { WoodA: Type-WoodA, 
WoodB: Type-WoodB, 
Catalyst: Type-Catalyst <Survives_As(Self)>>

Class O-Class

... Definition { PaperAB: Type-PaperAB,
PulpAB: Type-PulpAB,
Catalyst: Type-Catalyst }

Class Type-PaperAB

... Definition { Weight = 0.5 * (Weight(WoodA(GSource(Self)))
+ Weight(WoodB(GSource(Self))))
}

(The functions such as GSource, which are used to interpret the
specifications operationally in the ATN are explained further in chapter
6.). Apart from illustrating the use of the adjectives such as Irreversible
and Automatic, the example shows the following:

- **MtoN** signifies that some of the attributes of the source class may not expire.
- The **Survives_As** adjective associated with catalyst achieves this objective.
- The **GSource** function associated with the Weight attribute of the PaperAB indi-
cates that the weight of the paper produced is dependent in an evaluated way on
the weight of the pulp that was sourced into the transformation (Not on any other
reaction that the paper might undergo in its life time).

### 4.8 Chapter Summary

Chapter 4 presents the main contributions of this thesis in terms of modeling primitives
for dynamic object behavior. We introduce a set of four essential modeling primitives classified
according to whether any new identities are created in the course of the changes, and
whether the previous status is preserved. We also introduce a classification of attribute types
motivated by dynamic considerations. Recognizing that in order for the language to be
useful, it should be extensible to capture domain-specific semantics, we introduce the means
to describe dynamic features in relation to the semantics of the essential primitives. We illus-
trate extensibility by example of a set of selected dynamic features such as reversibility.

Syntactic elegance of the modeling language is not our main concern in this thesis.
However, for completeness and ease of explanation of the computational formalism, we
illustrate the expression of both the essential dynamic devices and dynamic features in a data
definition language in the concluding section of this chapter. The interpretation of both
essential devices and dynamic features in computational terms is addressed in chapter 6.
Chapter 5 Integrated Modeling with D'DIALECT(O)

We have presented a set of modeling primitives for dynamic domains. We argue in this chapter that these primitives should be used for conceptualization and modeling alongside traditional static modeling primitives such as inheritance relationships, to improve information models for dynamic domains. In support of this argument, we present a small set of guidelines for integrated static/dynamic modeling. These guidelines help to associate attributes with the "correct" classes, and to define the "correct" set of classes given our knowledge of the dynamic behavior of a domain. The word "correct" appears in quotes because data modeling is inherently tied to the intuitive ease for the observer, making claims regarding modeling ease formally unjustifiable in the absence of strong notions about intuition.

These guidelines are discretionary, and thus weaker than relational normalization or grammatical e-production removal, since they do not imply changes to data model (as 1NF versus NNF does) or computational characteristics (as e-production does). However, they are analogous to normalization rules in the sense that certain normalizations are not crucial to correctness, but done at the discretion of the designer (as in the case of 3NF and BCNF) who must trade off the schema fragmentation against the semantic benefits.

We make two types of claims regarding the formalism proposed in chapters 4 through 7. Some are more concrete since they are based on the underlying computational model. These include the claims that our primitives are realizable in a formal computational model which is provably expressive, and that the efficiency of reasoning with dynamic domains can range from real-time (average or atomic) through exponential-time and semi-decidable. These claims are discussed in section 6.8 after introducing the computational model.

The claims regarding modeling ease are weaker and rely on subjective measures. We claim that the data modeling primitives of d'language are intuitively suitable for conceptual modeling of object-centered domains when compared to most existing schemes. In addition, they are capable of representing and reasoning with practical concerns which are largely ignored by existing schemes. This claim is formally unjustifiable because the majority of the reviewed modeling schemes are not based on formal computational models. They offer primitives to model dynamic behavior, but not a methodology to integrate static and dynamic
modeling, or a notion of goodness measure of the resulting models. Moreover, since common formal foundations are largely lacking, this justification is weak because of its inability to review all existing schemes.

The justification in support of the claim of our proposed formalism as a medium for deriving intuitively good dynamic models therefore uses examples to illustrate:

- Guidelines relating dynamic and static domain modeling, along with their value in a modeling situation taken from the literature.
- The use of a notion of goodness for dynamic domain models. A framework to define measures of goodness, and means to specify the evolution of domain models to maximize their utility, are discussed in chapter Chapter 8.
- A hypothesis regarding the notion of "intuitively good" in dynamic modeling.

5.1 D'Language - a Unified Static / Dynamic Modeling Language

D'dialect(O) can be used to conceptualize and model a dynamic domain in an information system. Modeling in terms of these (or any other) data modeling primitives is a process of iterative modification of an initial axiomatization according to a set of principles until it meets some criterion of intuitive optimality from the user's viewpoint. The principles may include static ones such as normalization, grammar-based ones such as non-determinacy removal, and a third class of principles combining static and dynamic modeling, which we introduce below.

5.1.1 Hybrid Modeling Techniques

We are particularly interested in rules of modeling which capture the relationships of static and dynamic modeling concepts. Specifically, we are interested in modeling guidelines which relate the existence of dynamic dependencies to the desirability of static constructs, and vice versa. These conventions address the same problems as normalization theory at the conceptual and schema level, namely the reduction of redundancy in the schema. In OO schemas for dynamic domains, redundancy can take the following forms:

- If two reactions sourced and targeted at distinct classes are essentially identical in terms of their pre- and post-conditions and the actions needed to effect the reactions, then information is being duplicated. This can lead to the update of one part, inadvertently omitting the other part (dynamic update inconsistency).
In object-oriented schemas which incorporate the relationship of abstraction (important in many dynamic domains) among objects and allomorphism, redundant information is often included in the schema. Abstraction provides the ability to view the same object at different degrees of detail. Attributes which are identical at two different levels of abstraction may be duplicated in the representation, so that an update to an abstract representation of an object is not reflected in the more concrete representation of the same object.

The effects of this redundancy, as in the case of relational database schemas [160], becomes evident in: potential inconsistency in schema or object insertion/deletion/update.

OO normalization theory for static (or dynamic) domains has not been studied at depth, and is not the focus of this thesis. We propose the following guidelines based on the treatment of dynamic and static relationships as dependencies (similar to FDs) between two states of existence of an object over time.

5.1.1.1 Hybrid Modeling Guideline 1

Assume C is a set of statically non-redundant classes \( \{c_1, c_2, \ldots, c_n\} \), and that there is a set of reactions \( R_i \) from source class \( c_{is} \) to target class \( c_{it} \), such that \( c_{is} \) and \( c_{it} \) belong to C. For every two reactions \( R_i \) and \( R_j \) such that \( \text{preconditions}(c_{is}, R_i) = \text{preconditions}(c_{js}, R_j) \):

1. define a new abstract class \( c_{(i,j)s} \) with attribute set consisting of those attributes of \( c_{is} \) (and \( c_{js} \)) which occur in \( \text{preconditions}(c_{is}, R_i) \).
2. include \( c_{(i,j)s} \) in C
3. make \( c_{is} \) and \( c_{js} \) subsets of \( c_{(i,j)s} \)
4. replace \( R_i \) and \( R_j \) with reactions \( R_{(i,j)i} \) and \( R_{(i,j)j} \), sourced at \( c_{(i,j)s} \) and targeted at \( c_{it} \) and \( c_{jt} \) respectively.

**FIGURE 5.1** Hybrid Modeling Guideline 1
This transformation encapsulates the attributes involved in the dynamic pre-conditions associated with two reactions in a single static abstraction. These dynamic dependencies are soft constraints, unlike some relational normalizations which impose hard constraints among attributes. For example, the first-normal form constraint is a hard constraint on relational databases. On the other hand, the fourth normal form constraint is soft in the sense that the user has a choice of balancing schema fragmentation against update integrity. This transformation is not a hard rule; whether or not \( c_{i,j} \) is a bonafide static abstraction is not dependent only upon the attributes of \( c_{i,j} \) being common to more than one reaction.

Hybrid guideline 1 results in the definition of an additional schema element \( c_{(i,j)} \) which encapsulates the dynamic determinants in a single abstraction. It may be changed, added to or deleted as a single schema unit, instead of having to deal with all the source classes such as \( c_i \) and \( c_j \).

Usually, this transformation is applicable in generalizing both the source and target classes of a set of reactions, as in figure FIGURE 5.2.

Guideline 1 above suggests the creation of a static abstraction along the generalization dimension as a result of dynamic considerations. A similar transformation along the aggregation hierarchy is suggested as a guideline for combined static-dynamic modeling.

**5.1.1.2 Hybrid Modeling Guideline 2**

Assume that a single class of objects \( C \) is represented as concrete classes at two different levels of abstraction, \( C_1 \) and \( C_2 \), and that reactions \( R_1 \) and \( R_2 \) are sourced at \( C_1 \) and \( C_2 \).
respectively. If A is a common set of attributes in both levels of abstraction, C1 and C2, and preconditions(C1, R1) = preconditions(C2, R2), are predicates on attributes belonging to A, and similar conditions apply to the target classes of R1 and R2, then the following transformation is suggested:

i. Define a class CA with the attribute set A
ii. Redefine C1 and C2
   - remove A from the definition of C1 and C2.
   - define CA to an aggregated subpart of both C1 and C2.
iii. Replace R1 and R2 with a reaction RA sourced at CA
iv. Effect the same transformations at the target of R1 and R2.

Pictorially, this transformation appears as in figure FIGURE 5.3. It captures the semantics that the crucial transformation is undergone by the subpart CA. The effects of that reaction ripple through to C1 and C2 by virtue of the fact that CA is a subpart of C1 and C2.

FROM

TO

![Diagram](image)

FIGURE 5.3 Hybrid Modeling Guideline 2

5.1.1.3 Benefits of the Guidelines 1 and 2

Modeling guidelines 1 and 2 tie the static model of the domain to the dynamic relationships in the world. The converse of this cause-effect chain is also important in arriving at a good model: the dynamic relationships in the domain should also be affected by the nature of the static relationships among the objects. While we intuitively accept the static model as a priori in some way, there are situations where the analyst can specifically change the model to reflect this a priori status.
Guideline 2 translates a static picture of the world into a change in the dynamics, since the introduction of CA as an aggregate subpart of C1 and C2 is motivated, not by dynamic concerns, but by the need to rationalize the structure which is common to two levels of abstraction.

Guideline 3 addresses a third simple case in the same category.

5.1.1.4 Hybrid Modeling Guideline 3

Assume a static abstraction C and a reaction R sourced at C (likewise, targeted at C). Assume further that C is normalized as per the rules of static normalization into two aggregate subparts C1 and C2, such that the conditions and actions associated with R are predicated on attributes of both C1 and C2. The following transformation may be useful (figure FIGURE 5.4):

An M:N reaction is defined with C1 and C2 as source aggregate subparts of an entity C12 with no other attributes.

If, however, the conditions and actions associated with R concern attributes of either C1 or C2 exclusively, then R should be sourced at C1 or C2, respectively.

Guidelines addressing the dynamic aspects of the combination of two or more static abstractions into a single abstraction are specific to each situation and difficult to generalize.

5.1.2 Other Well-understood Modeling Guidelines

Well-understood static data modeling principles (object-oriented and non-object-oriented) include normalization techniques including domain key normalization [160] and dependency normalization. These techniques ensure minimal dependencies among attributes of distinct entities, and many inter-dependencies among the attributes within entities. They lead to the articulation of relationship entities capturing existence dependencies
among other entities. Object-oriented modeling devices and techniques such as those illustrated in [136] yield models with desirable constraint localization properties.

The proposed computational model, d'foundation, is based on the theory of formal language grammars, allowing grammatical techniques to be applied to modeling. These are adaptations from standard techniques to manipulate formal language grammars [79], such as elimination of epsilon-productions from regular grammars, conversion of context sensitive grammars to context-free grammars with annotations, and deriving equivalent deterministic parsers for definite grammars.

None of these techniques is discussed in this thesis, since they are independently well-documented. However, we emphasize the fact that the hybrid guidelines proposed above do not conflict with these established principles, since they are applied separately, before or after the entities suggested by the hybrid guidelines are postulated.

5.2 Guidelines Applied to static-dynamic data modeling

We illustrate the use of the guidelines above in the context of the modeling of the dynamic domain example below, taken from the domain of the dynamic relationships among states of existence in an academic program. The example illustrates the object-oriented analysis of a domain where the dynamic relationships are prior to the static ones, and which is refined based on the guidelines already given above.

![Diagram](Figure 5.5)

**Beginning the Conceptual Design**

Generalization generates superclasses from subclasses, and transition generates target classes from source classes. It is interesting to note some other parallels between transition and generalization: both relationships are binary, and both are identity relations. That is, generalization maintains identity over states of an object which exist at a given point in time, whereas transition maintains identity over various aspects of an object which exist at different points in time. Thus, transition modeling is a natural adjunct to the high-level conceptual modeling activities of choosing object classes and structuring them into a generalization.
hierarchy. Integrating dynamic and static modeling in this fashion promotes consistency and completeness in the resulting schema.

**FIGURE 5.6** Adding Reapplications

To begin the example, we represent that applicants either become students or are rejected, and students become alumni or fail to complete their programs (see figure FIGURE 5.5).

**FIGURE 5.7** Generalizing the Dynamic Behavior

Similarly, considering such questions as "Do all applicants become either students or rejects?", and "Can someone who has left the school re-apply to it?" leads to figure FIGURE 5.6. Note that this intuitive transition modeling step has identified object classes (i.e., Invitees, Declineds, Accepted, and No Shows) that otherwise may have been missed at this high-level stage.

More importantly from the viewpoint of dynamic integrity maintenance, it is identified that the Alumnus \(\Rightarrow\) Applicant transition is an extension, indicating that an alumnus who reappplies is considered to be currently both an applicant and an alumnus. The other
"reapplication" transitions are evolutions, reflecting the view that a person who reapplies after being rejected, for example, is viewed as an applicant who was once a reject.

At this point the generalization hierarchy in figure FIGURE 5.7 is suggested by the static relationships among the relevant dynamic entities obtained from the analysis of change in the domain (some transition links have been omitted in order to keep the figure simple). Those classes whose instances share the property of being currently associated with one of the two graduate programs are generalized into the Active class, and those whose instances share the property of having been removed from the system, but share a common transition back into it, are generalized into the Inactive class, as per our proposed guidelines.

Note that the Active class is specialized into two orthogonal groups of subclasses: one based on the attribute program (i.e., MSc versus Ph.D.) and the other based on status (i.e., applicant, etc.). (Each group of subclasses is indicated in the schema diagram by an arc through their respective generalization links.) The world may be bifurcated first according to program, and subsequently according to status or vice versa, leading to different generalization hierarchies. Some choices regarding the preferred hierarchy ensue from the known patterns of evolution of the potential members of the classes.

![Diagram](image-url)

**FIGURE 5.8** Completing the High-Level Conceptual Design

For example, suppose that in our axiomatization of the student database, an applicant to the Ph.D. program may be invited to either the Ph.D. or the M.Sc. programs, but all invitees remain in the same program until leaving school, except for those Ph.D. students who are experiencing difficulties and are permitted to complete a Master's degree. We would thus like to see the following transition links:

- Invitee $\Rightarrow$ Accepted, Accepted $\Rightarrow$ Student, and Student $\Rightarrow$ Alumnus

In only one of the hierarchies generated by the alternate bifurcations (that is, the one first bifurcated according to status) would there be object classes representing the general
CHAPTER 5. INTEGRATED MODELING WITH D‘DIALECT(0)

entity types *Invitee, Accepted, Student and Alumnus*. Therefore, this is the preferred alternative. In figure FIGURE 5.8, the integrated generalization hierarchy induced by the orthogonal bifurcation introduced in FIGURE 5.7 is made explicit, with the required transition links added, as per the guidelines (the Inactive classes which are irrelevant to the present discussion have been left out of the diagram).

The example illustrates the usefulness of the reaction abstractions as high-level design tools, both for the purpose of identifying the relevant object classes and the structure of the generalization hierarchy, and for identifying those transitions for which transactions need to be provided. The final resulting classes may be further normalized according to relational normalization rules, and the grammar interpreting the dynamics may be transformed through grammatical techniques to other equivalent grammars. The interpretation of a model such as the above in grammatical terms is addressed in chapter 6.

5.3 Comparison with other approaches

We have illustrated the use of the modeling guidelines to model a dynamic domain above. In most cases, we anticipate that the primitives of d‘language lead to economical representations of domains in an information model. Our reasons for believing in the above originate in the fact that most existing formalisms are either ad-hoc in origin (modeled with a particular example in mind) or derived from well known static formalisms such as ER. The latter carry baggage which is not always relevant to the modeling of a dynamic domain - in keeping to the conventions of the static modeling formalism, dynamic aspects are incorporated by distorting static mechanisms. D‘language is distinguished in being motivated by combined static and dynamic modeling as well as in having firm formal foundations.

![Diagram](image)
We illustrate an example of our convictions by discussing an example which has been previously used as a benchmark in the literature. We use the film-processing environment from [107] to compare our model. In the film processing environment, a "photo material", which may be either a piece of film, a slide, or a "supplied" negative, is turned into a print. Prior to creating the print, a piece of film must be turned into an "original" negative, and a slide must be converted into a "inter" negative (converted slide). A supplied negative may be copied directly to create the print. [107] details how a slightly extended Data-Flow diagram (e.g., FIGURE 5.9) may be translated into an extended Entity-Relationship diagram (EERD) (e.g., FIGURE 5.10).

FIGURE 5.10 Film Processing EERD

(FIGURE 5.9 and FIGURE 5.10 adapted from the paper.) Processes in Data-Flow diagrams are translated into relationship sets (e.g., DEVELOP) or weak entity sets (e.g., CONVERT) in EERDs. Inputs to processes are shown in EERDs as thin arrows, and outputs by thick arrows. In the EERD the concept of preservation (termed reusability) is handled by a new type of cardinality (termed involvement-cardinality or Inv). An involvement-cardinality of one (Inv1) on inputs to a process indicates that the input is not preserved. A distinction is made between homogeneous generalization (ISA) versus heterogeneous generalization (ISA*), the difference being that instances of a homogeneous subclass can "migrate" to another homogeneous subclass of the same generic superclass. Thus a primitive form of transition is implemented, albeit in a somewhat fuzzy manner, since, for example, an original negative may migrate to being a supplied negative but not vice versa.

FIGURE 5.11 shows a conceptual schema representing the film processing environment using the abstractions of our model. We represent transformations (productions) by
heavy dashed arrows (with a slash) in the schema and thusly, \( \rightarrow (\sim/\rightarrow) \), in running text. The schema contains the following productions, Slide \(-/\rightarrow\) Conv Slide and Neg \(-/\rightarrow\) Print, and one transformation, Film \(\rightarrow\) Orig Neg. In addition, it contains one evolution, Orig Neg \(\Rightarrow\) Supp Neg.

\[\text{FIGURE 5.11} \quad \text{D'LANGUAGE Film Processing Schema}\]

The above comparison, seen alongside the modeling process, illustrates our contention that the transition and generation abstraction mechanisms are potentially more useful for directly modeling at a high level the dynamics of an application domain than the facilities offered by process modeling systems.

The main reason for the creation of excessive notation in the Markowitz example occurs because EERDs:

- force distinctions to be made, which are not relevant to dynamic domains. e.g. notions of homogenous subclasses, generic superclasses lead to duplication in what types of objects can migrate to what other types.

- fail to make relevant and crucial distinctions. e.g. no representation of process or activity, or notion of time, exists - it is a static scheme. involvement cardinality is a generalization of a more specific distinction needed between identity-preserving change and birth/death pairs.

Though necessarily through informal example, we believe that the examples and comparison illustrate the usability and superiority of d'language as a modeling tool.
5.4 Chapter Summary

This chapter constitutes a secondary contribution of this thesis. We propose that the primitives suggested to capture dynamic information are usable in conceptualizing schema and performing transformations on the schema, to produce a better schema. We thus propose a few guidelines. The nature of the guidelines (i.e. the conditions under which the transformations are suggested) make them soft guidelines in that failure to make these transformations does not violate the model-inherent constraints. This is similar to the fourth normal form conditions as opposed to the first normal form conditions or epsilon-elimination rules. Our guidelines do not comprise a complete set, in the same sense that further normalization rules may be added for relational and object-oriented models.

We also provide an example of modeling which incorporates the proposed primitives and point out the benefits of using our primitives.
Chapter 6  D'Foundation: A Computational Framework

There are two aspects to modeling complex static and dynamic entities - the structural and the semantic aspects. The static structural aspect describes the atomic entities and how they are related to each other, such as aggregation and containment. The dynamic structure describes the topology of evolution histories in terms of the dynamic relationships among object-states, such as M:N relationships of creation, deletion and change in the actors of the domain. The semantic aspect adds meaning to the topologically related constituents. Examples of semantic content include precedence, legality, spatial containment and abstraction. Both structural and semantic contents of a model should be interpretable computationally.

In d'dialect(o) we have introduced means to specify complex dynamic relationships. In this chapter, we propose a computational framework for the structural and semantic aspects of dynamic (as per d’dialect(o) ) specifications in order to interpret operations based on specifications in d’dialect(o). The proposed framework models complex dynamic relationships among complex (object-oriented) static states. Our proposal uses sentential forms produced by dynamic grammars motivated by formal language theory for modeling dynamic structures.

This computational framework dynamic foundation (d'foundation), is an original adaptation of formal language methods for modeling open-ended, growing structures defined by rules in information systems. A dynamic grammar constrains the set of valid dynamic structures. Where grammars for languages define derivation trees and sentential forms, dynamic grammars define derivation histories and stored histories through successive application of its production rules. A stored history represents a temporal slice in a dynamic domain (or adaptive representation). A derivation history aids inference about related stored histories and helps to realize versions.

6.1  Computational Basis: Augmented Transition Networks

D’dialect is a user-interface language which provides a particular modeling framework, one of many sets of primitives which may be used to describe dynamic domains. Such an interface must be backed by a computational model implemented in some technology. In this
thesis, we do not address the technology, which may be a formalism such as Prolog. We address the question of providing a good computational model which is provably powerful as an expressive medium and also computable. A number of formalisms such as Petri Nets, embedded finite automata and different ad-hoc means (section 2.4) have been used in the past for specific tasks. D’foundation is based on Augmented Transition Network grammars (ATNs). A description of ATNs, including key definitions, expressive and computational characteristics, is included in appendix C. ATNs are attractive for the following reasons:

- **ATNs as a dynamic modeling tool** offers an object-oriented (versus process-oriented) view on the domain because of the primacy of states. This is advantageous in data modeling situations where the primary reference of user queries are the objects that participate in changes, not the processes of change.

- **ATNs offer a framework capable of inference about the static (e.g. aggregation and inheritance related) as well as dynamic attributes of an object in a single computational framework**.

- **The ATN framework is extensible** to capture flavors of these static and dynamic relationships (such as the semantics of spontaneous reactions).

- **ATNs afford the opportunity to combine schema and instance reasoning in a single framework**, leading to the ability to realize schema relativization.

- **The primary benefit of ATNs** is the ability to reason about equivalence and interchangeability of processes with respect to process goals. This falls out of the view of change descriptions as sentences generated in a formal language, and the large body of work on formal equivalence, containment etc. of formal languages.

Grammars provide a particularly good basis for dynamic structural and semantic modeling in information systems, apart from their advantages over other formalisms, because:

- **Databases are constructive (algebraic) to generate query execution plans. Grammars combine a of linear or branching structures in terms of the sentences and derivation trees with a set of constructive operations to generate and parse them.**

- **Grammars derive both linear structures (sentences) and derivation trees; thus both 'histories' and 'version hierarchies', two of the most common structures of temporal information systems, are amenable to grammatical description.**

- **Grammatical linear and tree-like structures may be interpreted as static structures with semantic contents such as inheritance and aggregation hierarchies of different kinds, and linear structures such as lists, all needed in information systems.**

The above characteristics place ATNs above logic-based systems with declarative semantics but search-based engines, such as the situation calculus. Within the domain of
grammars, ATNs are most suited to our goals because of the ease in translating d'dialect semantics to ATN. ATN’s are Turing-complete because of registers which allow non-local effects and because the functions defining, checking and constraining register values [13] are potentially unrestricted., as are some other formalisms. But the ATN notation provides additional advantages for expressing computations:

- ATN states and transitions map well onto d’dialect primitives.
- Dynamic structural relationships can be abstracted in the form of ATN links, which define the type of grammar represented by the ATN. Static and dynamic semantics are hidden in the register types, and associated conditions and actions.
- The above isolates these two separable issues and allows extensibility of structural and semantic aspects in mutual independence, as follows:
  - The ability to specify history-sensitive (resp. history-ignorant) derivation of future object-states, or monotonically-increasing (resp. unrestricted) histories is abstracted only in the grammar. Expressive power and computational complexity in this aspect is a function only of the underlying grammar.
  - The ability to specify whether an attribute of an object-state is derived through temporal reasoning from a previous object state, and whether one object state can be aggregated from other object-states, is a function of the attribute types and semantics in terms of conditions and actions on them.

These can be independently adjusted in ATNs. For instance, history-sensitive derivations and unrestricted histories may be disallowed, while aggregated object-states may be allowed. This would bias the framework towards allowing M:N reactions in the context of a relatively simple historical structure, with a certain computational complexity. Complexity of reasoning with dynamic structures can be isolated from semantics if:

- the alphabet of the register values is disjoint from the alphabet of the language describing dynamic behavior, and
- predicates on registers should not be dynamically added or removed

Under these restrictions, ATNs have a direct correspondence with attribute grammars (see appendix C).

### 6.1.1 Grammatically Defined Structures and Registers

The structural characteristics of histories is determined by the expressive power of the grammar implied by the ATN. ATNs are expressively capable of generating a variety of
topological structures denoting patterns of change through time (section 6.6). Key to the interpretation of these structures as inheritance hierarchies in one instance, and version trees in another, is the semantics associated with the nodes and attributes of the structure. In ATNs, the semantics associated with the structures are captured by registers associated with the nodes, and the pre- and post-conditions and actions defined in terms of the registers. In our proposal, nodes denote object-states; the nodal registers therefore determine the static and dynamic semantics of the object-states of the tokens (objects) resident at the node.

The structure and granularity of the contents of a register depend on the application. For example, a complex object-state may require registers that are set-valued. Not all registers must be involved in a dynamic operation. The complexity of the register values impacts the cost of operations, as does the complexity of the grammatical structure, as shown later.

**Definition 6.1** A register is a typed named variable for semantic information (called the value of the register) associated with a node of an ATN. They have the following characteristics:

1. Registers may hold complex structures such as trees
2. Values of registers associated with a node A may be assigned to other registers associated with a distinct node B by ATN actions.
3. Values of registers may be tested for specific conditions by the pre- and post-conditions associated with links of the ATN.

**Notation:** We refer to the registers associated with a node as \text{Registers}(Node), and distinguish between Inherited Reg(Node) and Synthesized Reg(Node).

The figure 6.1 illustrates some possible register assignments which may be made in the course of the traversal of an ATN link.

![Figure 6.1](Image)

**Figure 6.1** Register Arithmetic in ATN Traversals

Figure 6.1 depicts the situation where a register (reg1) associated with the containing ATN A is downloaded by the contained ATN C into a local register (reg3). The register value is
modified and propagated to the target node while traversing the local link labeled "e" as reg4. Later, the modified reg4 is uploaded into a local register of the containing ATN and propagated to the target node of the link labeled C (as reg5) while traversing ATN A. We introduce a special register Save_Set with every node, as follows.

**Definition 6.2** \( \text{Save}\_\text{Set}(o) \) is a distinguished register associated with every node of the ATN instance appropriated by object \( o \). It is always downloaded from the \( \text{Save}\_\text{Set}(o) \) register of the containing ATN when a contained ATN is traversed, always uploaded from the contained ATN when traversal of a contained ATN is complete, and always propagated within an ATN.

**6.1.1.1 ATN Semantics**

The computational semantics of ATN primitives are crucial to the interpretation of the ATN framework as a specification of dynamic behavior. In this context, we choose the following among the possible interpretations of specific ATN components:

- **Visibility of registers**: Registers(\( \text{Node} \)) are visible within the scope of the entire parse subtree rooted at \( \text{Node} \) if they are declared public. By default, they are local to \( \text{Node} \).

- **Actions**: Actions associated with an ATN link are instantaneous and atomic. There is no intermediate ATN node which manifests a subset of the effects of the actions associated with a link joining two given nodes.

- **Preconditions**: Preconditions are logical predicates defined on local registers which are not propagated, downloaded or uploaded from the given node with which they are associated. Preconditions are TRUE or FALSE only within the context of a node, i.e. when considering the possibility of a reaction.

- **Postconditions**: ATN postconditions are logical predicates defined on global registers defined at the root node (thus being visible to the whole network). In interpreting dynamic domains, we define post-conditions on local registers that must be satisfiable only locally. We limit global post-conditions through predicates on registers propagated from source to target node (and uploaded to containing ATNs) through the Save_Set. Save_Set contents may be modified over time.

- **Non-determinism**: Parsing and generation semantics of ATNs are non-deterministic. This implies that both in the course of generating a sentential form using an ATN description, and in the course of recognizing a sentence using an ATN description, the ATN generator or parser may be in more than one state (token may be in more than one node) simultaneously.

Su's work [155] characterizes the boundaries of dynamic specifications formally. The motivation behind dynamic and adaptive information systems is the specifiability of practical yet formally analyzable dynamic systems in the realm between and including the
extremes defined by Su’s SL and CSL(+). We believe that a majority of the dynamic behaviors of interest to us have context sensitive characteristics. Hence, we propose a framework of generators and recognizers for attribute grammars as the computational framework for our ATNs. However, two aspects of context free or context-sensitive grammars used to describe dynamic and adaptive behaviors are significant.

- First, non-terminal and terminal symbols can not be distinguished. A stored history may be a record of membership in class A, or alternatively, in a sequence of classes B,C,... as an elaboration of the membership in class A in terms of membership in A’s subclasses during the same interval. Thus A may be a terminal or non-terminal symbol. This indistinguishability does not increase complexity; it is a key feature of Developmental (Genetic) Grammars, which are provably equivalent to Indexed Languages [79].

- Second, since object-states and reactions are inter-related, a stored history may be viewed in two ways: as a sequence of reactions, and as a sequence of object states. While in the ATN notation, we visualize the reactions as links between states, we choose to visualize an object history as a sequence of object states, for obvious reasons. The choice of viewpoint does not affect complexity, just as a sentence in a CFL may be equivalently denoted by the sequence number of the productions that generated it.

### 6.2 Complexity of ATN Descriptions

We hypothesize that a significant number of dynamic patterns of interest in practical applications fall in the expressive range between regular and r.e. sets. This hypothesis is based on our study of applications in two areas (production planning and fault recovery). It is also intuitively clear that in scenarios such as product lifecycle modeling, future changes in an object’s characteristics is often determined by the size and semantics of their history. This predisposes the grammars describing such lifecycles to be inherently context-sensitive, and outside the set of context-free grammars. In some simpler cases, however, context free grammars are sufficient to describe the dynamic behavior. However, this hypothesis is inherently unprovable in the absence of a comprehensive study of all applications; counter-examples are readily constructed.

Expressive power comes with its associated computational penalty. Here we introduce the complexity considerations which are relevant to dynamics. The key consideration in designing frameworks for dynamic data schema is to allow trade-offs between the expressiveness (of dynamic structural relationships, the semantics of dynamic attributes and static structures as discussed in section 6.1) and the complexity of operations on dynamic entities.
6.2.1 Variables and Complexity Requirements

Computational complexity is expressed as a function of the complexity of the input(s). The measures of input complexity which are relevant to different operations in dynamic domains include:

- The *length of the history* of the object, i.e. the record of its membership in various classes over time.
- The *size of an object* in terms of *number of attributes*, which may or may not be independent of the length of its history.
- For class-based operations, the size of the class extension.

Operations (query and update) in dynamic and adaptive information systems include:

- The derivation of (a set of) attribute values for an individual object at a particular time, possibly from statically or temporally related object-states. Such operations should naturally have an acceptable complexity in terms of the size of the objects and their history.
- Selecting a set of objects which obey some condition at a particular time index. These operations should have acceptable complexity in terms of number of query attributes, sizes of the classes and their histories.
- Finding a time index when a (set of) object(s) satisfy some condition. These operations should have acceptable complexity in terms of all three variables.

Undecidable computations are universally unacceptable. In the case of schema, semidecidable computations may be acceptable in some cases, since we anticipate schema changes to be mediated by a human administrator who can choose to avoid certain schema change paths that are not promising. In most cases, we compare the complexity of operations in our framework to that of temporal databases, bearing in mind that the additional functionality extracts a cost. In a few cases, average complexity over a set of computations is relevant in dynamic and adaptive information systems, since the first computation is costly, but sets up the domain for subsequent inexpensive operations.

We do not impose specific bounds on expressiveness or complexity. Instead, we show the ATN formalism's flexibility for executing various d'language specifications by imposing limitations on dynamic structures or semantics. This is done by putting limits on the integrity constraints and actions associated with ATN, which improve the complexity of computations in d'foundation by not using the full power of unrestricted ATNs. Of course, such restrictions are reflected in d'dialect. For example, the conditions associated with a reaction...
could be restricted to equality checks based on atomic values, as specifiable in a WHERE clause of a temporal relational query language, excluding set or aggregate functions. Similarly, functions on registers could be restricted to exclude iterative or recursive functions. We justify such limitations on the basis of intuitions about the nature of typical reactions.

6.3 Interpreting D'dialect in D'foundation

In appendix C, we show the expressiveness and complexity concerns in using ATNs as the computational model for d'language. We need to show:

- How d'language(0) is interpreted computationally by d'foundation.
- The algorithms associated with the ATN computational model of d'foundation.
- Rules to map adjectives onto the computational devices of d'foundation.
- The complexity of computations in d'foundation.

We recall that d'alpha and the essential definitions are likely to be stable across most applications, but that the dynamic features may be expected to vary significantly. Therefore, we propose one-to-one mappings from essential dynamic devices onto d'foundation in section 6.3.2, and a methodology to map adjectives onto d'foundation in section 6.4. We use our methodology to map the adjectives introduced in section 4.6.1 onto d'foundation.

6.3.1 Basic Computational Interpretation in D'foundation

First, we propose interpretations for the basic static definitional devices, viz. identities, objects, attributes, object classes, utility and intervals in terms of our ATN computational formalism, d'foundation.

6.3.1.1 Static alphabet interpretations

An ATN grammar defines a set of sentences which may be generated or recognized as per the rules specified by the grammar. In our interpretation, each ATN token represents a dynamic object, which appropriates an instance of the ATN definition of valid object migrations, and migrates through a sequence of states during its lifetime, according to the rules defined by this ATN. The ATN instance describing the dynamic behavior of the object is used to derive the temporally determined attributes of the object through the generation and parsing algorithms associated with the ATN formalism. The following mappings constitute the core interpretation of d'language(0)'s static devices.
Interpretation 1: \((I (Id(O) = To))\) Object Identity is interpreted computationally by an ATN token associated with an instance of an ATN. ATN tokens are unique by definition. Each token which traverses an ATN, whether in a generative or parsing mode, therefore refers to a unique set of registers. (Hereafter, object and token are used interchangeably.)

Interpretation 2: \((I (Class) = n_{Class})\) The membership of an object in a class is interpreted computationally by the act of traversing an ATN link. However, for notational convenience, we label the target node of the link traversed with the name of the class in which the object belongs during the course of traversal of the link (figure 6.2). In this sense, Object classes are interpreted computationally by ATN nodes.

Interpretation 3: \((I (Class:a(O)) = n_{Class}:R_a(To))\) Definitional and factual attributes associated with a class are interpreted by ATN registers associated with the node representing the class. Attributes of a particular object (resp. token) as a member of a particular class (resp. node) are represented by registers associated with the node in the ATN instance appropriated by the token. I.e. \(n_{Class}:R_a(To) = X\) denotes the proposition \(Class:a(O) = X\). By the closed world assumption, the absence of a proposition asserts its negation.

Interpretation 4: \((I (Start(I) \text{ s.t. } O \in \text{Members-of}(Class)) \text{ exactly }) = n_{Class}:Begin(CMInt_I)(To); I (End(I) \text{ s.t. } O \in \text{Members-of}(Class) \text{ I }) = n_{Class}:End(CMInt_I)(To)\) The interval when an object is a member of Class (resp. interval during which a token is resident at a node) is interpreted by an implied interval CMInt_I. System-defined registers Begin(CMInt_I) and End(CMInt_I) interpret the start and end of CMInt_I. If a token \(To\) travels a link \(I\) from node \(A\) to node \(B\) at time \(t\), actions associated with \(I\) include \(B:End(CMInt_I)(To) \leftarrow A:End(CMInt_I)(To)\). \(A:End(CMInt_I)(To)\) and \(B:Begin(CMInt_I)(To)\) are then initialized to \(t\), so that \(I\) immediately precedes \(I'\). CMInt_I is implemented directly in our proposed data structures.

Interpretation 5: An object-state during a particular interval \(I\) is interpreted by the set of registers associated with the token corresponding to the object during the corresponding CMInt_I.

Interpretation 6: The set of tokens resident at a particular node \(n_{Class}\) at an instance of a particular ATN interprets the extension of Class. The conjunction of the propositions asserted by
the registers associated with a node \( n \) during a particular interval must satisfy \( \text{Defn}('\text{Class}') \) by implication of the definition of classes.

**Interpretation 7**: \( I (\text{Utility}(O, \text{Class}, T)) = n_{\text{Class}}; \text{Util}(T_O) \text{ s.t. } n_{\text{Class}}; \text{Begin}(\text{CMInt}_I)(T_O) \leq T \land n_{\text{Class}}; \text{End}(\text{CMInt}_I)(T_O) \geq T \) The utility of an object \( o \) as a member of a class \( C \) at time \( t \) is interpreted by a system-defined register \( \text{Util} \) associated with \( T_o \) attached to \( n_C \).

### 6.3.2 Structural Interpretation of Essential Dynamic Devices

Given the above interpretations of the static devices of d'language\((O)\), the dynamic relationships among the static entities can be interpreted in terms of the constructs of ATNs as follows. Essential definitions include two dynamic primitives for modeling the transition of objects, **evolution** and **extension**. These are mapped onto the ATN networks of d'foundation as follows in figure 6.3:

- **Object Evolution** \( [\Rightarrow \text{link}] \) from class \( A \) to class \( B \) is modeled computationally by an ATN subnetwork consisting of states \( n_A \) and \( n_B \), with one link \( <n_A, n_B> \) (sourced at \( n_A \) and targeted at \( n_B \)).

- **Object Extension** \( [=\Rightarrow \text{link}] \) from class \( C \) to class \( D \) is modeled computationally by an ATN subnetwork consisting of:
  - Two states \( n_C \) and \( n_D \).
  - Two links \( <n_C, n_C> \) (sourced and targeted at \( n_C \)) and \( <n_C, n_D> \) (sourced at \( n_C \) and targeted at \( n_D \)).
  - The **Concurrent Branch** (a set of Preconditions and Actions) associating \( <n_C, n_C> \) and \( <n_C, n_D> \) (See section 6.3.3.3).

- The two link traversals above are encapsulated in an **atomic transaction** as a single link \( <n_C, n_D> \).

**Transitions**

![Diagram of ATN Equivalents](attachment:image.png)

**ATN Equivalents**

- **Concurrent Branch**

**FIGURE 6.3** ATN Interpretation of Transition Links

The ATN links modeling extension and evolution are labeled with the name of the cor-
responding transition, and carry the conditions and actions determined by the translation semantics in following sections. Similarly, the generative production and transformation abstractions are interpreted as follows in figure 6.4:

**Generations**

![Diagram of Generations]

**ATN Equivalents**

![Diagram of ATN Equivalents]

- **Object Transformation** [--> link] of an object in class A to another in class B is modeled in d'foundation by two nodes \( n_A \) and \( n_B \), a distinguished third node ",", and by two links \( <n_A,> \) and \( <>,n_B> \).
- link \( <n_A,> \) is an ATN link.
- link \( <>,n_B> \) is also an ATN link, with the **Token Identification Rules** associated with it.

The two link traversals above are encapsulated in an *atomic transaction* as a single link \( <n_A,n_B> \). The class "," is a special *null class* whose members are historical (The definitions of these actions and classes are given later in the thesis).

- **Object Production** [/-/> link] of an object in class D from another object in class C is modeled by two ATN nodes \( n_C \) and \( n_D \), a distinguished node ",", and three links
  - links \( <n_C,n_C> \) and \( <n_C,> \) are associated with the **Concurrent Branch Precondition and Action**.
  - link \( <>,n_D> \) has the **Token Identification Rules** associated with it.

These three link traversals above are encapsulated in an *atomic transaction* as a single link \( <n_C,n_D> \). The semantic equivalence of these translations is derivable from the definitions of the primitives in d’dialect and the semantics of ATN links. This is outlined in sections 6.3.3.

### 6.3.3 Interpretation of Semantics for Essential Dynamic Devices

The preceding sections gave the structural interpretation of the essential dynamic primitives in terms of ATN nodes and links. The interpretation must also enforce:
compliance of the reacting object(s) with the intensional definitions of source and target classes before and after the reaction, respectively.

- constraints such as Token Identification and Concurrency, as in section 6.3.
- constraints imposed in the period after \( t \) by a reaction undergone at time \( t \).

### 6.3.3.1 Candidature, Link Selection and Basic Pre-conditions & Actions

Suppose that a token \( T_0 \) is resident at a node \( n_{\text{Class}} \). \( T_0 \) becomes eligible for traversing a link \( R = <n_{\text{Class}}, T> \) if:

1. a register \( n_{\text{Class}}:r(T_0) \) is updated at \( t \); and
2. \( O \) ceases to satisfy the definition associated with Class, s.t. \( \text{Class} \in \text{Collections}(O) \), and
   \[ [Z, t] (i.e. \text{if } N = \land (\text{Class} : a(O) = X) \text{ for all registers } n_{\text{Class}} : R_a(T_0), \text{ according to interpretation } 3, \text{ and } \neg \lambda x(\text{Defn(Class)})(N) | t) \]

Alternatively, it becomes eligible if a distinguished register \( n_{\text{Class}}:\text{Candidate}(T_0) \) is manually set to 'True' in order to trigger a change explicitly. \( n:\text{Candidate}(T_0) \) is set to false by the ATN engine immediately after a token is considered for a link traversal, whether or not the traversal actually executes. These rules control computations by making a token eligible for a reaction if and only when an external stimulus. The following is the default pre-condition:

\[ (n_{\text{Class}}:\text{Candidate}(T_0) = \text{True}) \lor (n_{\text{Class}}:a(T_0) < - \text{Val} | t \land \neg \lambda x(\text{Defn(Class)})(O) | t) \] on every ATN link \( <n_{\text{Class}}, X> \), which may be modified by specific reaction types and adjectives.

After these conditions are met, there may be two or more links which can be potentially traversed. To choose among such a set of links, the following heuristic is proposed.

3. if \( L_i \in \text{Outlinks}(n_{\text{Class}}) \), and \( O \) satisfies a subset \( S_i \) of the definitional conditions of \( \text{Target-Class}(L_i) \), and \( S_j < S_k \) for some \( j,k \) in \( 1..n \), then \( S_k \) is preferred over \( S_j \) as a candidate reaction for object \( O \) to traverse. Other heuristics may also be used.

If \( O \) traverses link \( R \) at \( t \), it is updated so that it satisfies the definition of the target node after \( t \). \( \lambda x(\text{Defn(Target(R))})(O) | [t, \text{now}] \), which is the default post-condition. This implies:

- Evaluate(R), for every \( R(T_0) \) such that \( n_{\text{Class}}:R(T_0) \) is defined, where \( n_{\text{Class}} = \text{Target-Class}(L) \) for the traversed link \( L \).

The constraints under which each register value is evaluated depends on the attribute and reaction type, as discussed later. If it is defined temporally or on some static hierarchy, the temporal or static semantic rules are used to evaluate it. By default, if the register is defined on both the source and target classes of the traversed link, the value is propagated. The conditions above specify that if an event at \( t \) alters an objects state so that it ceases to be a
member of a class which it belonged to before \( t \), then it becomes eligible to traverse the link to a node whose associated class definition it may satisfy if some updates are made to it. Becoming eligible for a reaction does not mean that it \textit{must} take place.

Reaction semantics themselves are determined by concurrent branching and identification action semantics, preservation of status (extensions and productions preserve the status of the object before the reaction, while evolutions and transformations don’t), whether the reaction is qualified by during, etc. We address these subjects in the following sections.

\subsection*{6.3.3.2 Membership Specialization (During)}

Evolutions may be qualified by the \textit{during} keyword (section 4.5), which carries the implication that the interval associated with the source object-state is specialized in terms of other object-states. Note the distinction from the semantics associated with status preserving reactions (extension and production) where objects have two concurrent object-states after the reaction. In the example in chapter 4, the intension is that as far as the sales order is concerned, it should be suspended within the validity interval of the original sales order, not as a reaction in the future relative to the sales-order object-state. A part of the ATN interpreting these semantics follows in figure 6.5. The interpretation is as follows: suppose \(<n_A, n_B>\) is qualified by \textit{during}, \( n_A: \text{Begin}(CMInt_l)(T) = t_{\text{Start}} \) and \( n_A: \text{End}(CMInt_l)(T) = t_{\text{End}} \) where \( CMInt_l \) refers to the interval of residence of a token \( T \) at \( n_A \). Suppose the link \(<n_A, n_B>\) is traversed. Then the following actions replace the default for the Begin and End registers.

\( n_B: \text{Begin}(CMInt_l)(T) \leftarrow t_{\text{Start}} \) and \( n_B: \text{End}(CMInt_l)(T) \leftarrow t_{\text{End}} \) as a constructor. Subsequently, \( n_A: \text{End}(CMInt_l)(T) \leftarrow \phi \) and \( n_A: \text{Begin}(CMInt_l)(T) \leftarrow \phi \). All registers of \( n_B \) are evaluated.
according to the interpretation of the link $\langle n_A, n_B \rangle$. These ensure that the token $T$ lives at $n_B$ within the interval of residence at $n_A$, obeying the definition of the class $B$.

### 6.3.3.3 Concurrent branching

In representing dynamic behaviors through ATNs, the process of reaching a goal state from a starting state, or the process of generating a representation with desirable properties from a starting representation, is a process of finding a parse tree for a sentence containing the end-state as well as the starting state. This may proceed concurrently along two or more paths. Concurrent branching implies a logical replication of the generator into more than one copy, along with the state at the instant of the branch. This is followed by distinct state progression by each of the copies. With the support of the data structures in our proposed formalism (chapter 7), the concurrent partial parse trees can be represented in parallel through the implicit duplication of the token representing the object.

When a token (object) resident at node (class) $A$ concurrently traverses a pair of ATN links to nodes $B$ and $C$ (where $B$ or $C$ is identical to $A$), the semantics demand that states $B$ and $C$ be inhabited at once, only once, and at the same time. I.e., the token:

1. Must be a member of both $B$ and $C$ at the same time as a result of the event; and
2. Should not imply multiple concurrent membership in either of the classes $B$ or $C$ as a result of the single event of traversing the concurrent branch.

![Concurrent Branch](image)

**FIGURE 6.6** Concurrent Branching in the Grammar

Condition (1) is satisfied if the object satisfies the intensional definition of both $B$ and $C$ in any time interval immediately following the time of the traversal. This is predicated on the condition that $\text{Defn}(B) \land \text{Defn}(C)$ is satisfiable, which is a pre-runtime check.

Condition (2) specifies that the object can not "become" a member of $B$ (or $C$) at time instant $t$ if it already was a member of $B$ (or $C$) in an interval immediately preceding $t$. This condition is specifically relevant in the case where $B$ (or $C$) is identical to $A$, as in figure 6.3. In this case, d'foundation needs to ensure that the object does not become a member of $A$ at $t$,
but retains its previous membership. We interpret conditions 1 and 2 as below:

**Definition 6.3 Concurrent Branch Rules**

1. Assertions $n_{\text{class}}(r_{a}(o) = x | i_{1} = [t_{1}, t_{2})$ and $n_{\text{class}}(r_{a}(o) = x | i_{2} = [t_{3}, t_{4})$, s.t. $i_{1} = \text{l-prec}(i_{2})$ or Overlaps($i_{1}, i_{2}$) or Covers($i_{1}, i_{2}$), together imply $n_{\text{class}}(r_{a}(o) = x | i_{3} = [t_{5}, t_{6})$, where $t_{5} = \text{min}(t_{1}, t_{3})$ and $t_{6} = \text{max}(t_{2}, t_{4})$.

2. If $t$ is the time of the reaction and $i_{1}$ is the interval immediately preceding $t$, and at any node, $\text{CMInt}_{t}(T_{o}) | i_{1} = \text{l-prec}(\text{CMInt}_{t}(T_{o}) | i_{1})$, then Begin($\text{CMInt}_{t}(T_{o})$) is defined as Begin($\text{CMInt}_{t}(T_{o}) | i_{1}$) and End($\text{CMInt}_{t}(T_{o})$) is defined as End($\text{CMInt}_{t}(T_{o}) | i_{1}$).

3. Modify the standard pre-condition for eligibility for link traversal to $(n_{\text{class}} \cdot \text{Candidate}(T_{o}) = \text{True}) \lor (n_{\text{class}} \cdot a(T_{o}) < Val \cup i)$.

Rule (1) ensures that when a register associated with a token at a node evaluates to its previous value after traversing a link, no discontinuity is imposed on its value interval. Rule (2) ensures condition (2), which specifies that if a token resides at the same node before and after a link traversal, the CMInt's corresponding to the two periods of residence are concatenated, making the period of residence at the node a single long interval. Rule (3) ensures that if the source class definition is preserved after attribute update as it must for extensions and productions, it does not eliminate the links corresponding to the reaction from being eligible. These rules ensure that concurrent links will be traversed even if one link is sourced and targeted at the same node, and that membership intervals at such nodes will not be broken into two discrete intervals by the time point of the reaction.

**6.3.3.4 Token Identification Rules**

When a reaction results in the creation of an object (identity) distinct from the objects antecedent to the reaction, a token identification action should occur. Token identification implements the semantics of transferring the properties of an object in one class to a distinct object, and is interpreted using the identity creation function:

**Definition 6.4 Token Identification Rules**

If $S = \text{Source\_Class}(R)$, and $T = \text{Target\_Class}(R)$ of the link $R$ traversed at time $t$, then:

1. $\text{Id}(T_{\text{new}}) = \text{Create\_Identity}(\text{Id}(T_{o}))$, where $T_{\text{new}}$ is a new token.
2. $(\forall R \in \text{Registers}(n_{S})$, if $(n_{S} \cdot R(T_{o}) = X) \mid [a, b]$) then do:
   - evaluate $n_{T} \cdot R(T_{\text{new}}) \mid [t, b]$, and
   - (for transformations only) $n_{S} \cdot R(T_{o}) = X \mid [a, t]$

Token identification consists of creating a new token and identity (using either a generator or constructor function - see reversibility), and transferring all the registers of the source
object to the newly-created identity. For transformations, the previous attributes are capped
at t, and T₀ does not traverse to node nₜ, logically making O impossible to reference after t.

6.3.3.5 Reactions that do not preserve previous status
Reactions that do not preserve the status prior to the reaction event are evolutions and
transformations. In the case of evolutions where the source and target objects are identical,
the target class definition must not imply the source class definition. Failure of this condition
would imply membership in both the source and target classes after the reaction, violating
the definition of evolutions. This restriction does not apply to transformations, where source
and target object instances are distinct. The pre-runtime condition to be ensured is thus:

∀R ∈ {Evolution} [ Defn(Target_Class(R)) ⊨ Defn(Source_Class(R)) ]

This does not ensure there is no object state o such that λxDefn(Target_Class(R))(o) ∧
λx(Defn(Source_Class(R)))(o), where λx denotes the abstraction of the attributes in the defi-
nition. This condition is enforced by the post-conditions associated with the link I(R) inter-
preting R (for transformations A-->B, I(R) denotes the link <nₐ>,), as follows:

(A) ¬(∧ₐ Condition(Source_Class(I(R)), rₐ(T₀)) \ t, a ∈ Defn_Attr(Source_Class(I(R))).
Applicable to evolutions, it may be satisfied as a pre-condition (section 6.3.3.1, condition 2).

(B) (∧ₐ Condition(Target_Class(I(R)), rₐ(T₀)) \ t, a ∈ Defn_Attr(Target_Class(I(R))). This
condition is applicable to both evolutions and transformations.

(A) says that the conjunction of the conditions on the definitional registers of the source
of the ATN link interpreting the reaction should not be satisfied by the register values inter-
preting the definitional attributes of the source class. (B) specifies that the conjunction of the
conditions on the definitional registers of the target of the ATN link interpreting the reaction
should be satisfied by the registers interpreting the definitional attributes of the target class.

Evaluate actions associated with the links enable satisfaction of the conditions above, as
per the rules given in section 6.3.3.7.

6.3.3.6 Reactions that preserve previous status
Reactions which preserve the status of the source object-state comprise extensions and
productions. For extensions, the definitions of source and target classes must not be mutually
contradictory and therefore simultaneously unsatisfiable. This condition does not normally
apply to productions, where source and target object-states refer to distinct objects. The fol-
lowing satisfiability condition must therefore be true before runtime.
\[ \forall R \in \{\text{Extension}\} \vdash [\text{Defn}(\text{Target Class}(R)) \land \text{Defn}(\text{Source Class}(R)) ] \]

This does not ensure that all object states \( o \) satisfy \( \lambda x \text{Defn}(\text{Target Class}(R))(o) \land \lambda x (\text{Defn}(\text{Source Class}(R)))(o) \), where \( \lambda x \) denotes the abstraction of the definitional attributes. This is ensured by the post-conditions associated with the link \( I(R) \) interpreting \( R \) (for productions \( A \to B \), \( I(R) \) is the link \( \langle n_A, \cdot \rangle \), for extensions \( A =/=> B \), it is the link \( \langle n_A, n_B \rangle \), as follows:

(A) \( (\land_a \text{Condition}(\text{Source Class}(I(R)), r_a(T_o))) \mid _t, a \in \text{Defn Attr}(\text{Source Class}(I(R))) \).

Applicable to evolutions, it may be satisfied as a pre-condition (section 6.3.3.1, condition 2).

(B) \( (\land_a \text{Condition}(\text{Target Class}(I(R)), r_a(T_x))) \mid _t, a \in \text{Defn Attr}(\text{Target Class}(I(R))) \).

This condition is applicable to both evolutions and transformations.

(A) says that the conjunction of the conditions on the definitional registers of the source of the ATN link interpreting the reaction should be satisfied by the register values interpreting the definitional attributes of the source class. (B) specifies that the conjunction of the conditions on the definitional registers of the target of the ATN link interpreting the reaction should be satisfied by the registers interpreting the definitional attributes of the target class. For extensions \( x = o \), for productions, \( x \) is the virtual member of the null class 1.

Evaluate actions associated with the links enable satisfaction of the conditions above, as per the rules given in section 6.3.3.7.

6.3.3.7 Rules for Attribute Evaluation

We elaborate on the evaluate function on attributes mentioned in the previous sections as follows, discussing their adequacy in section 6.3.4. The registers which interpret attributes associated with the source and target nodes of a reaction are evaluated under the constraints shown in figure 6.7. There are nine types of attributes which are treated differently. These are labeled A through I in the figure, and behave as follows when a link is traversed:

- **Type (A)** - definitional in both source and target classes. Registers of this type are treated differently depending on the type of reaction. In evolutions, \( n_{\text{Source}}:r_a(T_o) \) is mandatorily verified and evaluated under Condition(Target, a); at least on register is optionally reevaluated under the constraint \( \neg \text{Condition}(\text{Source}, a) \) if no register of Type (B) or (C) is similarly evaluated. For extensions, \( n_{\text{Source}}:r_a(T_o) \) is mandatorily verified and evaluated under Condition(Target, a) \land Condition(Source, a). For transitions, \( n_{\text{Source}}:r_a(T_o) = n_{\text{Target}}:r_a(T_o) \); for productions, \( n_{\text{Source}}:r_a(T_o) \) is verified and evaluated under Condition(Source, a); it may not be propagated to \( n_{\text{Target}}:r_a(T_o) \), which is verified and evaluated under Condition(Target, a); for transformations, only \( n_{\text{Target}}:r_a(T_o) \) is significant under Condition(Target, a).

- **Type (B)** - definitional attributes of the source class, which are factual attributes of
the target. For evolutions, \( n_{\text{Source}}:r_{a}(T_{0}) \) is optionally evaluated under the constraint \( \neg\text{Condition}(\text{Source}, a) \) if no register of Type (A) or (C) is similarly updated. For extensions and productions, register \( n_{\text{Source}}:r_{a}(T_{0}) \) is verified if \( r_{a} \) is updated at \( t \), and optionally evaluated under \( \text{Condition}(\text{Source}, a) \). For evolutions, \( n_{\text{Source}}:r_{a}(T_{0}) \) may not have the same value as \( n_{\text{Target}}:r_{a}(T_{0}') \), which may be optionally evaluated for all generations.

<table>
<thead>
<tr>
<th>Target</th>
<th>Definition</th>
<th>Factual</th>
<th>Not defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Factual</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>Not defined</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
</tbody>
</table>

**FIGURE 6.7** Registers Categorized by Constraints Applicable

- **Type (C)** - definitional attributes \( a \) of the source class which are not defined on the target class. Registers \( n_{\text{Source}}:r_{a}(T_{0}) \) are verified if \( r_{a} \) is updated at \( t \), and optionally evaluated under \( \text{Condition}(\text{Source}, a) \) for extensions and productions. For evolutions, \( n_{\text{Source}}:r_{a}(T_{0}) \) is optionally evaluated under the constraint \( \neg\text{Condition}(\text{Source}, a) \) if no register of Type (A) or (B) is similarly updated.

- **Type (D)** - factual attributes \( a \) of the source class, which are definitional with respect to the target class. The interpreting registers are mandatorily verified and optionally evaluated under the constraint \( \text{Condition}(\text{Target}, a) \). For transitions, \( n_{\text{Source}}:r_{a}(T_{0}) = n_{\text{Target}}:r_{a}(T_{0}) \); for productions, \( n_{\text{Source}}:r_{a}(T_{0}) \) may not have the value of \( n_{\text{Target}}:r_{a}(T_{0}') \); for transformations, only \( n_{\text{Target}}:r_{a}(T_{0}') \) is significant.

- **Type (E)** - factual attributes \( a \) shared by source and target classes. Registers interpreting these attributes are optionally evaluated, under no constraints. For transitions, \( n_{\text{Source}}:r_{a}(T_{0}) = n_{\text{Target}}:r_{a}(T_{0}) \); for productions, \( n_{\text{Source}}:r_{a}(T_{0}) \) may not have the value of \( n_{\text{Target}}:r_{a}(T_{0}') \); for transformations, only \( n_{\text{Target}}:r_{a}(T_{0}') \) is significant.

- **Type (F)** - factual attributes of the source class that are not defined in the target class. The registers \( n_{\text{Source}}:r_{a}(T_{0}) \) are optionally evaluated under no constraints for extensions and productions.

- **Type (G)** - definitional attributes \( a \) of the target class undefined in the source. Registers \( n_{\text{Target}}:r_{a}(T_{0}) \) are mandatorily evaluated under \( \text{Condition}(\text{Target}, a) \).

- **Type (H)** - factual attributes \( a \) of the target class, undefined in the source class. Registers \( n_{\text{Target}}:r_{a}(T_{0}) \) may be optionally evaluated, under no constraints.

- **Type (I)** - Not applicable to source or target classes
6.3.4 Integrity of Interpretation in D'foundation

The interpretation of reactions in terms of d’foundation preserves their semantics, as per the following reasoning. For each reaction type, we interpret the source class in ATN terms, traverse the links which interpret the reaction type, and correlate the resulting ATN state with the target object-state, thereby arguing for the integrity of interpretation. Note that this is a reasoning for soundness only. Completeness is not an issue because the interpretation of reactions in ATNs cover a subset of the linguistic space of unrestricted grammars, which ATNs are equivalent to. We illustrate with reference to figure 6.8, which shows the ATN interpretation of the d’dialect specification in figure 3.6.

**Evolution:** If an object O is a member of a class S during an interval I immediately prior to time t, when it undergoes the evolution S ==> T, then the interpretation of this reaction as given in section 6.3 ensures that at t, O is not a member of S and is a member of T.

**Informal Proof:** (i) The initial conditions for evolutions specify S ∈ Collections( O) in an interval I= [t1, t2), which implies O satisfies λx.Defn(S) during I by definition 4.2, and T ∉ Collections(O) during I implies O does not satisfy λx.Defn(T) during I.

(ii) By interpretation 6 in section 6.3.1, there exists a token T0 corresponding to O (interpretation 1) at a node nS corresponding to class S (interpretation 2). Attributes S:a(O) are interpreted by registers nS: R_a(T0)(interpretation 3). Since O satisfies λx.Defn(S), which is a conjunction of conditions on the definitional attributes of S, v_a satisfies Condition(S, a), where nS:R_a(T0)= v_a for all R_a interpreting a ∈ Defn_Attr(S).

(iii) Interval I= [t1, t2) is interpreted by the register values nS:Begin(CMIntI)(T0) = t1, and nS:End(CMIntI)(T0) = t2.

(iv) R = S ==> T is interpreted by the single ATN link <nS,nT> as described in section
6.3.2. ATN semantics allow the link \(<n_S, n_T>\) to be traversed if the pre-conditions are satisfied and the state and actions satisfy the constraints imposed by the post-conditions, as follows:

- **Precondition**: \((n_S: \text{Candidate}(T_o) = \text{True}) \lor (n_S: a(T_o) <- \text{Val}_t \land \neg \lambda x(\text{Defn}(S))(O) \mid t))\) is satisfied. This may happen by setting \text{Candidate} to True explicitly, or when an attribute is updated violating the definition of \(S\).

- The following actions are carried out:
  1. \(T_o\) goes from node \(n_S\) to node \(n_T\) at time \(t\), leaving the extension of \(S\) and joining \(T\)
  2. By interpretation 4 in section 6.3.1, the interval registers are updated as follows:
     \(n_S: \text{End}(\text{CMInt}_I)(T_o) \leftarrow t\), \(n_T: \text{Begin}(\text{CMInt}_T)(T_o) \leftarrow t\) and \(n_T: \text{End}(\text{CMInt}_T)(T_o) \leftarrow t_{\text{now}}\) by default, unless the reaction is qualified by the keyword *during* (section 6.3.3.2).
  3. Registers of type (A) through (I) are mandatorily or optionally verified and evaluated under the rules given in section 6.3.3.7.
  4. No other rules apply.

  (v) The constraints on the evaluation of registers of type (A), (B), and (C) ensure that at least one register \(n_{\text{Source}}: r_a(T_o)\) corresponding to attribute \(a \in \text{Defn}_{\text{Attr}}(S)\) is evaluated at time \(t\) under the constraint \(\neg \text{Condition}(S, a)\) (note: this may be satisfied as a pre-condition).
  Thus \(S \in \text{Collections}(O) \mid t\) as \(\land_a n_{\text{Source}}: r_a(T_o) = X\) does not imply \(\neg \lambda x(\text{Defn}(\text{Source}(R)))(O) \mid t\).
  
  (vi) The constraints on evaluation of registers of type (A), (D), and (G) ensure that all registers \(n_{\text{Target}}: r_a(T_o)\) corresponding to attribute \(a \in \text{Defn}_{\text{Attr}}(T)\) are evaluated at time \(t\) under the constraint \(\text{Condition}(T, a)\). Under interpretation 3 in section 6.3.1 and definition 4.1, \(\land_a \text{Condition}(T, a), \text{for } a \in \text{Defn}_{\text{Attr}}(T)\) implies \(\lambda x(\text{Defn}(\text{Target}(R)))(O) \mid t\).
  
  (vii) Conditions (v) and (vi) can be simultaneously satisfied because of the a priori condition imposed on the definitions of \(S\) and \(T\) in section 6.3.3.5. Even if attributes of type (B), (C), (D) and (G) do not exist, there is at least one interpretation of an attribute of type (A) which satisfies \(\neg \text{Condition}(S, a) \land \text{Condition}(T, a)\).
  
  (viii) (v) and (vi) imply the postcondition attached to \(R\), viz. \(\neg \lambda x(\text{Defn}(\text{Source}(R)))(O) \mid t \land \lambda x(\text{Defn}(\text{Target}(R)))(X) \mid t\), which satisfies the definition of evolution. These post-conditions are valid only at time-point \(t\) because they are defined on local registers (not uploaded, downloaded or propagated).
  
  (ix) By the discussion in section 4.3.1.3 and interpretation 7 of section 6.3.1, the utility of \(O\) as a member of \(S\) is 0 and as a member of \(T\) is 1.

**Extension**: If an object \(O\) is a member of a class \(S\) during an interval \(I\) immediately prior
to time \( t \), when it undergoes the extension \( S = \Rightarrow T \), then the interpretation of this reaction as given in section 6.3 ensures that at \( t \), \( O \) is a member of \( S \) and of \( T \).

**Informal Proof**: (i) The initial conditions for evolutions specify \( S \in \text{Collections}(O) \) in an interval \( I = [t_1,t_2) \), which implies \( O \) satisfies \( \lambda x.\text{Defn}(S) \) during \( I \) by definition 4.2, and \( T \not\in \text{Collections}(O) \) during \( I \) implies \( O \) does not satisfy \( \lambda x.\text{Defn}(T) \) during \( I \).

(ii) By interpretation 6 in section 6.3.1, there exists a token \( T_\circ \) corresponding to \( O \) (interpretation 1) at a node \( n_\circ \) corresponding to class \( S \) (interpretation 2). Attributes \( S:a(O) \) are interpreted by registers \( n_\circ:R_a(T_\circ) \) (interpretation 3). Since \( O \) satisfies \( \lambda x.\text{Defn}(S) \), which is a conjunction of conditions on the definitional attributes of \( S \), \( v_a \) satisfies \( \text{Condition}(S,a) \), where \( n_\circ:R_a(T_\circ) = v_a \) for all \( R_a \) interpreting \( a \in \text{Defn\_Attr}(S) \).

(iii) Interval \( I = [t_1,t_2) \) is interpreted by the register values \( n_\circ:\text{Begin}(\text{CMInt}_I)(T_\circ) = t_1 \), and \( n_\circ:\text{End}(\text{CMInt}_I)(T_\circ) = t_2 \).

(iv) \( R = S = \Rightarrow T \) is interpreted by the ATN links \( \langle n_\circ, n_T \rangle \) and \( \langle n_\circ, n_\circ \rangle \) encapsulated in a single transaction as described in section 6.3.2, and linked by the Concurrent Branch conditions of section 6.3.3.3. ATN semantics cause links \( \langle n_\circ, n_T \rangle \) and \( \langle n_\circ, n_\circ \rangle \) to be traversed if their pre-conditions are satisfied and the resulting states satisfy the post-conditions:

- Precondition \( \left( (n_\circ:\text{Candidate}(T_\circ) = \text{True}) \lor (n_\circ:a(T_\circ) \leftarrow \text{Val1}_t) \right) \) may be satisfied by setting \( \text{Candidate} \) to True explicitly, or when an attribute is updated. Notice that the update need not violate the definition of \( S \) due to point 3 in definition 6.3.
- The following actions are carried out:

1. \( T_\circ \) goes from node \( n_\circ \) to node \( n_T \) and \( n_\circ \) at time \( t \), residing in the extensions of \( S \) and \( T \)
2. By interpretation 4 in section 6.3.1 for the link \( \langle n_\circ, n_T \rangle \), the interval registers are updated as follows: \( n_\circ:\text{End}(\text{CMInt}_I)(T_\circ) \leftarrow t, n_T:\text{Begin}(\text{CMInt}_I)(T_\circ) \leftarrow t \) and \( n_T:\text{End}(\text{CMInt}_I)(T_\circ) \leftarrow t_{\text{now}} \) unless the reaction is qualified by \( \text{during} \) (section 6.3.3.2).
3. By interpretation 4 in section 6.3.1 for the link \( \langle n_\circ, n_\circ \rangle \), the interval registers are updated as follows: \( n_\circ:\text{End}(\text{CMInt}_I)(T_\circ) \leftarrow t, n_\circ:\text{Begin}(\text{CMInt}_I)(T_\circ) \leftarrow t \) and \( n_\circ:\text{End}(\text{CMInt}_I)(T_\circ) \leftarrow t_{\text{now}} \). But applying condition 2 of definition 6.3 to this update, we get \( n_\circ:\text{Begin}(\text{CMInt}_I)(T_\circ) \leftarrow n_\circ:\text{Begin}(\text{CMInt}_I)(T_\circ) \) and \( n_\circ:\text{End}(\text{CMInt}_I)(T_\circ) \leftarrow t_{\text{now}} \).
4. Registers of type (A) through (I) are mandatorily or optionally verified and evaluated under the rules given in section 6.3.3.7, under the conditions of point 1 in definition 6.3.

(v) The constraints on the evaluation of registers of type (A), (B), and (C) ensure that all registers \( n_\circ:r_a(T_\circ) \) corresponding to attribute \( a \in \text{Defn\_Attr}(S) \) are evaluated at time \( t \) under the constraint \( \text{Condition}(S,a) \) (note: this may be satisfied as a pre-condition). Thus \( S \in \text{Collec-
(vi) The constraints on evaluation of registers of type (A), (D), and (G) ensure that all registers $n_T::r_a(T_o)$ corresponding to attribute $a \in \text{Defn}\_\text{Attr}(T)$ are evaluated at time $t$ under the constraint $\text{Condition}(T, a)$. Under interpretation 3 in section 6.3.1 and definition 4.1, $\wedge_a \text{Condition}(T, a)$, for $a \in \text{Defn}\_\text{Attr}(T)$ implies $\lambda x(\text{Defn}(\text{Target}(R)))(O) \mid_t$

(vii) Conditions (v) and (vi) can be simultaneously satisfied because of the a priori condition imposed on the definitions of $S$ and $T$ in section 6.3.3.6; there is at least one interpretation of all attributes of type (A) which satisfies $\text{Condition}(S, a) \wedge \text{Condition}(T, a)$.

(viii) (v) implies the postcondition attached to $<n_S,n_T>$, viz. $\lambda x(\text{Defn}(\text{Source}(R)))(O) \mid_t$ is satisfied, and (vi) implies $\lambda x(\text{Defn}(\text{Target}(R)))(X) \mid_t$ is satisfied for $<n_S,n_T>$. Together, these satisfy the definition of extension. These post-conditions are valid only at time-point $t$ because they are defined on local registers (not uploaded, downloaded or propagated).

(ix) By the discussion in section 4.3.1.3 and interpretation 7 of section 6.3.1, the utility of $O$ as a member of $S$ is 1 and as a member of $T$ is also 1. Until the next attribute update occurs, or Candidate is manually set, the same reaction will not be considered again.

**Transformation:** If an object $O$ is a member of a class $S$ during an interval $I$ immediately prior to time $t$, when it undergoes the transformation $S \rightarrow T$, then the mapping in section 6.3 implies that a new object legally related to $O \in T$ while $O$ itself has no existence at and after $t$.

**Informal Proof:** The interpretation of productions in ATNs combines sequentially the interpretation of an evolution $<n_S,.>$ and a token creation link $<.,n_T>$ within a single transaction. We first adapt the proof for evolutions to the case of transformations, and later extend it with the token creation semantics.

**Step 1 : Evolution $<n_S,.>$**

(i) The initial conditions for transformations specify $S \in \text{Collections}(O)$ in an interval $I=[t_1,t_2)$, which implies $O$ satisfies $\lambda x.\text{Defn}(S)$ during $I$ by definition 4.2. By interpretation 6 in section 6.3.1, there exists a token $T_o$ corresponding to $O$ (interpretation 1) at a node $n_S$ corresponding to class $S$ (interpretation 2). Attributes $S::a(O)$ are interpreted by registers $n_S$: $R_a(T_o)$ (interpretation 3). Since $O$ satisfies $\lambda x.\text{Defn}(S)$, $v_a$ satisfies $\text{Condition}(S, a)$, where $n_S::R_a(T_o)=v_a$ for all $R_a$ interpreting $a \in \text{Defn}\_\text{Attr}(S)$. Intervals are treated as in evolutions.

(ii) The single ATN link $<n_S,.>$ is traversed if the pre-conditions are satisfied and the state and actions satisfy the constraints imposed by the post-conditions, as follows:

- Precondition $((n_S::\text{Candidate}(T_o) = \text{True}) \lor (n_S::a(T_o) \leftarrow \text{Val} \mid_t \land \neg \lambda x(\text{Defn}(S))(O) \mid_t))$
is satisfied. This may happen by setting Candidate to True explicitly, or when an attribute is updated violating the definition of S.

- The following actions are carried out:

1. \( T_o \) goes from node \( n_S \) to \( \cdot \) at time \( t \), leaving \( S \) and joining the null class.

2. By interpretation 4 in section 6.3.1, the interval registers are updated as follows:
   \( n_S:End(CMInt_I)(T_o) < t \) , \( l:Begin(CMInt_I)(T_o) < t \) and \( l:End(CMInt_I)(T_o) < t_{now} \) by default, unless the reaction is qualified by the keyword during (section 6.3.3.2).

3. Registers of type (A) through (I) are mandatorily or optionally verified and evaluated under the rules given in section 6.3.3.7.

   (iii) Note that for transformations, it may be that \( S \in Collections(O) \mid t \) as \( \wedge_a l:r_a(T_o) = X \) may imply \( \lambda x(Defn(Source(<n_S,->)))(O) \mid t \)

   **Step 2: Token identity creation \(<,n_T>\)**

   The second link within the transaction is associated with the token identification rules of definition 6.4. A new token \( T_P \) with new identity are created via identity creation function.

   (iv) The constraints on evaluation of registers of type (A), (D), and (G) ensure that all registers \( n_r:T(T_o) \) corresponding to attribute \( a \in Defn_Attr(T) \) are evaluated at time \( t \) under the constraint \( Condition(T, a) \). Under interpretation 3 in section 6.3.1 and definition 4.1,
   \( \wedge_a Condition(T, a) \) , for \( a \in Defn_Attr(T) \) implies \( \lambda x(Defn(Target(R)))(O) \mid t \)

   (v) Since \( n_T:r_X(T_o) = \phi \) for all attributes \( X \), and \( End(i) = t \) \( \forall \) value intervals \( i \) of attributes \( Y \) \( n_S:r_Y(T_o) \), \( T_o \) and its attributes can not be accessed after \( t \) on the current ATN timeline.

   Computationally, \( O \) ceases to be accessible on the current timeline.

   (vi) All registers and their values evaluated as per definition 6.4. (iv), and the fact that \( n_T:r_a(T_P) = n_r:a(T_o) \) , \( \forall a \in Defn_Attr(T) \), implies \( \lambda x(Defn(Target(R)))(O) \mid t \)

   (vii) (v) and (vi) imply the postcondition attached to \( R \), \( \lambda x(Defn(Target(R)))(P) \mid t \), and \( Collections(O) = \phi \mid t \) which satisfies the definition of transformation. These post-conditions are valid only at time-point \( t \) because they are defined on local registers.

   (viii) By the discussion in section 4.3.1.3 and interpretation 7 of section 6.3.1, the utility of \( O \) as a member of \( S \) is \( \phi \) at \( t \) and that of \( P \) as a member of \( T \) is 1 at \( t \).

   \( T \in Collections(P) \) at \( t \). Moreover, \( P \) does not exist in any interval preceding \( t \), and \( O \) does not exist in any interval covering \( t \).

   **Production:** If an object \( O \) is a member of a class \( S \) during an interval \( I \) immediately prior to time \( t \), when it undergoes the production \( S -/-> T \), then the interpretation of this
reaction in section 6.3 ensures that at t, O remains a member of S and a new object is created and made a member of T.

Informal Proof Outline: The interpretation of productions in ATNs combines sequentially the interpretation of an extension <nS, > and a token creation <nT> within a single transaction. The reasoning mirrors the two-step process for transformations, substituting the reasoning for extensions in the place of the reasoning for evolutions in step 1. Note that point (vii) under the reasoning for extensions does not apply to productions since the corresponding pre-condition is not assured at modeling time. However, since register evaluation occurs in step 2, it does not affect the semantic equivalence of the interpretation.

6.3.5 Example of Reactions in Paper Production

We illustrate the interpretation through the traversal of selected links in figure 6.8. The computations involved in effecting some of these reactions in terms of ATNs are briefly discussed here, and again in relation to adjectives in section 6.4.2. Assume that the classes are defined simplistically as follows (all Conditions(C,A) are implied by the definitions):

- **Sales_Order**(SO): \( (Recv\_Date \neq \phi \land Deliv\_Date \leq 1/1/2000) \)
- **Susp_Sales_Order**(SSO): \( (Recv\_Date \neq \phi \land Deliv\_Date \geq 1/1/2000) \)
- **Matl_Order**(MO): \( (Matl\_Date \leq 1/1/2000) \) where \( Matl\_Date = Source.Deliv\_Date-10 \)
- **Goods_Req**(GR): \( (Item\_List \neq \phi) \) with a Factual.Att Req.Date = Source.Mat1_Date
- **Goods_Invoice**(GI): \( (Item\_Name \neq \phi \land Req\_Date \leq 12/1/1999) \)

Assuming that initially there is an object O with Id(O) = 12345, which is a member of the class SO since date 10/1/1991 till the current time tnow.

Evolution (During): The reaction from Sales_Order(SO) to Susp_Sales_Order(SSO) is an evolution during the interval of validity of the source state. The initial conditions are satisfied if \( Recv\_Date(O) = 10/1/1991 \) and \( Deliv\_Date(O) = 10/1/1995 \). The interpretation in ATNs is a token \( T_0 \) resident at node nSO, such that register values are \( n_{SO}:r_{Recv\_Date}(T_0) = 10/1/1991, n_{SO}:r_{Deliv\_Date}(T_0) = 10/1/1995, n_{SO}:Util(T_0) = 1, n_{SO}:Candidate(T_0) = False, n_{SO}:Begin(CMInt_i)(T_0) = 10/1/1991 \) and \( n_{SO}:End(CMInt_i)(T_0) = t_{now} \).

The reaction is modeled by ATN link \( <n_{SO}, n_{SSO}> \) with preconditions \( (n_{SO}:Candidate(T_0) = True \lor (\exists! \exists R \exists Val(n_{SO}:R(T_0) < Val \land (n_{SO}:r_{Recv\_Date}(T_0) = \phi \lor n_{SO}:r_{Deliv\_Date}(T_0) > 1/1/2000))); } \). This condition can be satisfied by updating one or more
of these registers. Suppose \( n_{SO}:\text{Candidate}(T_o) \leftarrow \text{True} \) at 6/21/1995 as an explicit trigger by the user. According to the rules, the following registers must be verified and if necessary updated under the constraints expressed as postconditions.

- **Type A** registers: \( r_{\text{Recv\_Date}}(T_o) \), and \( r_{\text{Deliv\_Date}}(T_o) \), applicable to \( n_{SO} \) and \( n_{SSO} \)

There are no other register types in this example. The constraints imposed by the post-conditions are

\[
(r_{\text{Recv\_Date}}(T_o) = \phi \lor r_{\text{Deliv\_Date}}(T_o) > 1/1/2000 \lor \) \( \land (r_{\text{Recv\_Date}}(T_o) \neq \phi \lor \land r_{\text{Deliv\_Date}}(T_o) \geq 1/1/2000 \lor \).
\]

The resulting constraints after simplification are \( r_{\text{Recv\_Date}}(T_o) \neq \phi \lor \land r_{\text{Deliv\_Date}}(T_o) > 1/1/2000 \lor \). Note that the satisfiability of this constraint is ensured by the precondition on the definitions of \( SO \) and \( SSO \). In our example, if \( r_{\text{Deliv\_Date}}(T_o) \leftarrow 1/2/2000 \lor \), and \( r_{\text{Deliv\_Date}}(T_o) \) value is propagated from \( n_{SO} \) to \( n_{SSO} \), then both are satisfied.

The associated actions consist of

\[
n_{SO}:\text{End}(\text{CMInt})(T_o) \leftarrow 6/21/1995 \lor 6/21/1995, \quad n_{SSO}:\text{Begin}(\text{CMInt})(T_o) \leftarrow 6/21/1995 \lor 6/21/1995, \quad n_{SSO}:\text{End}(\text{CMInt})(T_o) \leftarrow n_{SO}:\text{End}(\text{CMInt})(T_o)
\]

\( = t_{\text{now}} \lor 6/21/1995 \lor \), and the token \( T_o \) moves to node \( n_{SSO} \). Since the reaction is qualified by **during**, \( n_{SSO}:\text{Begin}(\text{CMInt})(T_o) \leftarrow n_{SO}:\text{Begin}(\text{CMInt})(T_o) \) and \( n_{SSO}:\text{End}(\text{CMInt})(T_o) \leftarrow n_{SO}:\text{End}(\text{CMInt})(T_o) \) as constructor attributes, to ensure that state \( SSO \) exists within the interval of validity of \( SO \). Util is evaluated, and Candidate reset, as expected during the traversal.

**Production**: The reaction from \( \text{Sales\_Order}(SO) \) to \( \text{Material\_Order}(MO) \) is a production.

The sales order continues to exist (for accounting) while the material order initiates manufacturing. The initial conditions are satisfied if \( \text{Recv\_Date}(O) = 10/1/1991 \) and \( \text{Deliv\_Date}(O) = 10/1/1995 \). The interpretation in ATNs is a token \( T_o \) resident at node \( n_{SO} \), such that register values are

\[
n_{SO}:r_{\text{Recv\_Date}}(T_o) = 10/1/1991, \quad n_{SO}:r_{\text{Deliv\_Date}}(T_o) = 10/1/1995, \quad n_{SO}:\text{Util}(T_o) = 1, \quad n_{SO}:\text{Candidate}(T_o) = \text{False}, \quad n_{SO}:\text{Begin}(\text{CMInt})(T_o) = 10/1/1991 \quad \text{and} \quad n_{SO}:\text{End}(\text{CMInt})(T_o) = t_{\text{now}}.
\]

The production is modeled by ATN links \( <n_{SO}, n_{SO}> \) and \( <n_{SO}, n_{MO}> \) associated with the concurrent branching rules, followed by the link \( <, n_{MO}> \) with the token identification rules. These are handled within a single transaction.

**Extending** \( <n_{SO}> \) : The preconditions on this traversal, according to our semantics, are

\[
\{n_{SO}:\text{Candidate}(T_o) = \text{True} \lor \exists \exists \exists \exists \text{Val} \ (n_{SO}:R(T_o) \leftarrow \text{Val} \lor \}
\]

noting that the update need not violate the definition of \( SO \). Suppose \( n_{SO}:\text{Candidate}(T_o) \leftarrow \text{True} \) at 9/21/1995 as a result of explicit triggering by the user. The following registers must be verified and if necessary updated under the constraints expressed as postconditions.

- **Type C** registers: \( n_{SO}:r_{\text{Recv\_Date}}(T_o) \), and \( n_{SO}:r_{\text{Deliv\_Date}}(T_o) \)
Type G registers: \( n_{MO}:r_{Matl-Date}(T_o') \)

There are no other register types in this example. The constraints imposed by the post-conditions are \( n_{SO}:r_{Recv-Date}(T_o') \neq \phi \mid t \land n_{SO}:r_{Deliv-Date}(T_o) \geq 1/1/2000 \mid t \) \land \( n_i:r_{Matl-Date} \leq 1/1/2000 \mid t \). The resulting constraints after simplification are \( r_{Recv-Date}(T_o) \neq \phi \mid t \) and \( r_{Deliv-Date}(T_o) \leq 1/1/2000 \mid t \) at this point. The satisfiability of this condition is assured by propagating \( r_{Recv-Date}(T_o) \) and \( r_{Deliv-Date}(T_o) \) values from \( n_{SO} \) to \( n_{SO} \) and \( t \), since no type A registers affecting the definitions of \( SO \) and \( MO \) exist in our example. Interval registers are updated as follows: \( n_{SO}:End(CMI_{Int})(T_o) \leftarrow 9/21/1995 \quad 9/21/1995 \), \( n_{SO}:Begin(CMI_{Int})(T_o) \leftarrow 9/21/1995 \quad 9/21/1995 \), \( n_{SO}:End(CMI_{Int})(T_o) \leftarrow t_{now} \quad 9/21/1995 \), \( n_{SO}:Begin(CMI_{Int})(T_o) \leftarrow 9/21/1995 \quad 9/21/1995 \), \( n_i:End(CMI_{Int})(T_o) \leftarrow t_{now} \quad 9/21/1995 \). As per the concurrent branching rules, \( CMI_{Int}I \) and \( CMI_{Int}I' \) are combined into \( CMI_{Int}Y \) at \( n_{SO} \) by the following rules:

\( n_{SO}:Begin(CMI_{Int}Y)(T_o) \leftarrow n_{SO}:Begin(CMI_{Int})(T_o) = 10/1/1991 \) \land \( n_{SO}:End(CMI_{Int})(T_o) < t_{now} \). The token \( T_o \) resides at both \( n_{SO} \) and \( t \). With respect to node \( n_{SO} \), therefore, the registers \( n_{SO}:Begin(CMI_{Int})(T_o) = 10/1/1991 \), \( n_{SO}:End(CMI_{Int})(T_o) = t_{now} \), \( n_{SO}:r_{Recv-Date}(T_o) = 10/1/1991 \) \land \( n_{SO}:r_{Deliv-Date}(T_o) = 10/1/1995 \), along with the presence of \( T_o \) at \( n_{SO} \) imply an unbroken membership in \( SO \), as per the desired semantics of productions.

Reaction \( \leftarrow n_{MO} \): To at node \( n_t \) has 3 registers \( n_i:r_{Recv-Date}(T_o) \), \( n_i:r_{Deliv-Date}(T_o) \) and \( n_i:r_{Matl-Date}(T_o) \). The value of \( n_i:r_{Matl-Date}(T_o) \) is successfully evaluated by the rule \( n_{SO}:r_{Deliv-Date} = 10 \) under the constraint \( n_i:r_{Matl-Date}(T_o) \leq 1/1/2000 \mid t \) to the value 9/21/1995. A new token \( T_o' \) corresponding to a new object \( O' \) with identity \( Id(O') = f(Id(O)) = f(12345) \) is created. \( f \) may be a constructor such as \( f(x) = x+10000 \) which can be reversed, or an evaluator such as \( f(x) = crypt(x)*10000 \) which may not be reversible. Since \( Matl-Date \) is the only attribute of class \( MO \), the register assignments are done as follows: \( n_{MO}:r_{Matl-Date}(T_o') \leftarrow n_i:r_{Matl-Date}(T_o), n_{MO}:Begin(CMI_{Int}')(T_o') \leftarrow n_i:Begin(CMI_{Int}')(T_o) \) \land \( n_{MO}:End(CMI_{Int}')(T_o') \leftarrow n_i:End(CMI_{Int}')(T_o) \). The token \( T_o' \) is moved to node \( n_{MO} \). These assignments and the presence of the token imply a new object with membership in \( MO \) starting at 9/21/1995 and spanning \( t_{now} \). This completes the interpretation of \( SO-\rightarrow MO \).

Extensions, such as the reaction from \( Goods_Requisition(GR) \) to \( Goods_Invoice(GI) \), and transformations such as the reaction from \( Material_Order(MO) \) to \( Goods_Requisition(GR) \) are interpreted similarly. Since the logic is implied by the above examples, we omit detailed explanations of these types.
6.4 Interpreting Adjectives in D'foundation

Section 6.3 shows the interpretation of the essential dynamic primitives in terms of the d'foundation (ATN) primitives. In practical applications, a small set of dynamic features (adjectives, section 4.6) specific to the application domain often carry the most vital semantic information. Examples include spontaneity in process industry information support systems and irreversibility (distinguished from the ability to rollback) in financial applications. Without loss of generality, we assume in this section that reactions relate a single source object to a single target object. Section 6.4.2 illustrates how the concept is extensible to M:N reactions.

Augmented transition networks provide pre-conditions, actions, and post-conditions in terms of which the semantics of adjectives can be specified. If the semantics of a dynamic feature can be expressed as logical conditions on the reacting object and related objects before, after and during the reaction, they can be interpreted by ATN pre- and post-conditions and actions interpretable by the ATN engine. This is a one-time modeling task for a system analyst not repeated at runtime, for which we discuss a methodology in the following sections.

6.4.1 Limitations on Mapping Adjectives to ATN Constructs

Spontaneity, irreversibility and M:N exemplify common adjectives qualifying reactions in practice, but we admit the possibility of other adjectives being crucial to the modeling of specific domains. For example, an useful adjective qualifying evolutions in the maintenance management of telecommunications ia balanced (section 6.4.2.4). Balanced evolutions control changes of the object in terms of the number of times it undergoes the qualified evolution in relation to other state-changes. For example, evolution of circuit cards from in-service to standby can be qualified as balanced, to control the proportion of occasions when cards are stood by when in service. An extensible language for modeling dynamic behavior should incorporate means to specify and interpret the semantics of such adjectives consistently.

The interpretation of each dynamic relationship in terms of the immediate ATN topology is fixed by the rules in section 6.3. The overall ATN topology is thus pre-determined by the modeling of reactions to and from specific classes, and not modifiable directly by the semantics of an adjective. We restrict the expression of adjective semantics as follows:

- adding, modifying or deleting non-domain-specific registers (those that do not interpret domain attributes), including Save_Set, Begin and End. Adjectives can add such registers to Registers(Source(L)) and Registers(Target(L)) if L is the link.
adding meta-conditions on Pre_Conds(L), Post_Conds(L) and Actions(L) as complete formulae, and/or adding terms to them which refer to the non-domain specific registers.

If the adjective semantics extend beyond these changes, the modeler has to compensate during the initial modeling by adding more classes or links, as in the case of M:N reactions. The complexity of the conditions and actions obviously affects the cost of computation.

Guidelines to map such constraints onto ATN conditions and actions are proposed below, and illustrated in section 6.4.2. The semantics of a given adjective may be expressed as a constraint W (in temporal logic [164]), which implies secondary constraints on the object(s) involved in the relationship before, after, and during the reaction, as follows:

- **Domain-independent Meta-conditions on the temporally prior and subsequent object-states in the relationship R**, as denoted below:

  \[ [i] \land \text{Condition}_i(\text{Source-Class}(R), A)_{t-1}, [ii] \land \text{Condition}_j(\text{Target-Class}(R), B)_{t-1}, [iii] \land \text{Condition}_k(\text{Source-Class}(R), A)_{t+1} \]

  (where I immediately precedes t, A is a distinguished attribute such as Id or Begin(CMInt) of SourceClass(R), or meta-attributes such as Defn_Attrs() or Factual_Attrs(). B is the same for Target-Class(R). W \implies \text{Condition}_i, \text{Condition}_j, \text{Condition}_k)

- **Conditions on related object-states other than those in the dynamic relationship, prior to the point at which the relationship is established**: The related object-state may be statically (e.g. aggregation) or dynamically (e.g. generative source) related to the object-states involved in the reaction.

  \[ [iv] \land \text{Condition}_i(X, A), \text{Relation}(X, \text{Source-Class}(R))_{t-1} \]

  (where I is immediately prior to t, Relation denotes some relationship, W \implies \text{Condition}_i, and A is a distinguished attribute or meta-attribute of class X)

- **Legal relationships among individual source and target object-state attributes, which are not implied by the definitions of the source and target classes**:

  \[ [v] \land (\text{Condition}_{m}(\text{Source-Class}(R), A)_{t-1} \implies \text{Condition}_{n}(\text{Target-Class}(R), B)_{t+1}) \]

  (I is immediately before t, A is a distinguished attribute or meta-attribute of SourceClass(R), B is the same for Target-Class(R), and the implication above is implied by W).

- **Conditions on the temporally related object-states after the point when the relationship is established**: After a reaction, the distinguished attributes or meta-attributes of the source and/or target object-states may need to satisfy additional conditions.

  \[ [vi] \land \text{Condition}_p(\text{Source-Class}(R), A), \text{Condition}_q(\text{Target-Class}(R), B)_{t+X} \]

  (Condition, are implied by W, and A is a distinguished attribute or meta-attribute of the source object-state and B is the same for the target object-state)
Conditions on related objects after the point when the relationship is established:

\[ [\text{vii}] \land \text{Condition}_q(X, A), (\text{Relation}(X, \text{Source-Class}(R)) \lor \text{Relation}(X, \text{Target-Class}(R))) \mid_{(X)} \]

(Conditioni are implied by W, and A is a distinguished attribute or meta-attribute of the object-state X related to the source or target object-states statically or dynamically)

This thesis does not propose a theoretical model capable of proving the completeness of the conditions given above with respect to the needs of specifying the semantics of all possible adjectives. Such a proof is not possible in the absence of a model for all possible adjectives that may qualify a reaction. The claim of adequacy of these conditions for capturing the computational intent in the semantics of arbitrary adjectives is therefore weak. It is justified only on the grounds of experience with some common adjectives and the limitation that adjective semantics can be modeled only by modifying conditions and actions based on registers associated with the links which interpret the basic reactions.

These semantics can be interpreted in ATN terms by modifying (negating, qualifying or adding to) \( \text{Actions}(L), \text{Pre-conds}(L), \text{Post-Cons}(L) \) related to the link(s) \( L \) as follows:

- Categories of registers such as \( \text{Defn-Attr}(\text{Source}(L)) \), \( \text{Defn-Attr}(\text{Target}(L)) \) (as well as Fact_Attr, Dormant_Attr and others defined in chapters 3 and 4) can be addressed

- Domain-independent adjective semantics would not normally refer to domain-dependent registers or their specific values, but are not prohibited from doing so.

- System-defined registers \( \text{Util}, \text{Begin}(\text{CMIntI}), \text{End}(\text{CMIntI}), \text{Save-Set} \) and \( \text{Id} \) may be tested and set. Meta-conditions may be applied on functions manipulating them.

- New registers \( r \) may be created and initialized in order to capture semantic information related to the adjective, and constraints \( \text{Conditions}(r) \) may be specified on the values of the newly added registers.

Adjectives whose semantics can be expressed in these terms can be modeled under our framework. The mapping is a manual one-time task. The conditions [i] through [vii] given earlier can (not necessarily) affect ATN devices as follows (section 6.4.2 gives examples):

\[
\begin{align*}
\text{C}_1 &= \land \text{Condition}_1(\text{Source-Class}(L), A) \mid_1 \land \text{Condition}_1(X, A), \text{Relation}(X, \text{Source-Class}(L)) \mid_1 \land \text{Condition}_1(X, \text{Relation}(X, \text{Source-Class}(L))) \mid_1 \land \text{Condition}_1(X, \text{Relation}(X, \text{Source-Class}(L))) \mid_1 \land \\
\text{Pre-Cons}(L) &\leftarrow \text{Pre-Cons}(L) \land C_1 \\
\text{Preconditions}(L) \text{ must remain consistent (logically satisfiable).}
\end{align*}
\]

\[
\begin{align*}
\text{C}_2 &= \land \text{Condition}_1(\text{Target-Class}(L), B) \mid_1 \land \text{Condition}_1(\text{Source-Class}(L), A) \mid_1 \land \text{Condition}_1(\text{Target-Class}(L), B) \mid_1 \land \text{Condition}_1(\text{Target-Class}(L), B) \mid_1 \\
\text{[iii]} \text{ may be defined as post-conditions on local registers interpreting the}
\end{align*}
\]
attributes of Source_Class(L) and Target_Class(L).

\[ \text{Post}_\text{Conds}(L) \leftarrow \text{Post}_\text{Conds}(L) \land C_2 \]

Postconditions(L) must remain consistent. In satisfying postconditions Actions may be modified:

\[ \text{Actions}(L) \leftarrow \text{Actions}(L) \cup \text{Action} \]

- \( C_3 = \land (\text{Condition}_m(\text{Source}_\text{Class}(L), A) \mid t) \Rightarrow \text{Condition}_p(\text{Target}_\text{Class}(L), B) \mid t) \)

[v]: In the case when a condition on a target object is implied by another condition on the source object, postconditions on local registers may be modified thus:

\[ \text{Post}_\text{Conds}(L) \leftarrow \text{Post}_\text{Conds}(L) \land (\text{Condition}_m(\text{Source}_\text{Class}(L), A) \Rightarrow \text{Condition}_p(\text{Target}_\text{Class}(L), B)) \]

The conditions are defined on the local registers of Source_Class(L) and Target_Class(L).

Post_Conds(L) must remain consistent. In satisfying postconditions, Actions may be modified:

\[ \text{Actions}(L) \leftarrow \text{Actions}(L) \cup \text{Action} \]

Certain adjectives impose constraints applicable in the future on the objects in a reaction. Since these objects may establish dynamic relationships in the future with other objects, the conditions on the future object-states may be imposed through the use of post-conditions on registers propagated to future object-states through the Save_Set register (definition 6.2).

\[ C_4 = \land (\text{Condition}_p(\text{Source}_\text{Class}(L), r), \text{Condition}_p(\text{Target}_\text{Class}(L), r) \mid [t, X)) \]

[vi]: \( \land (\text{Condition}_q(X, r), (\text{Relation}(X, \text{Source}_\text{Class}(L)) \lor \text{Relation}(X, \text{Target}_\text{Class}(L)) \mid [t, X)) \]

[vii]: \([t, X)\) indicates the duration into the future with respect to the time of reaction \( t \) when the conditions are applicable. \( C_4 \) may be imposed through constraints on the appropriate registers. If the interval of validity is only \( t \), the constraints may be 'and'ed to the link post-conditions. If they extend into the future,
these registers may be added to Save_Set, so that they are propagated, uploaded and downloaded to future current object-states. (see figure 6.9).

\[
\text{Save}\_\text{Set} \leftarrow \text{Save}\_\text{Set} \cup r; \text{Conditions}(r) \leftarrow \text{Conditions}(r) \cup \text{Conditions}_y \mid_{[t,X]}, y = o,p \text{ or } q
\]

The conditions apply on the appropriate registers during the interval \([t,X)\) only.

6.4.2 Interpreting Specific Adjectives

We illustrate the modeling of three adjectives (spontaneous evolutions, M:N reactions, and reversible reactions introduced in section 4.6.1) which are found to be useful in production planning and maintenance management in terms of the discussion earlier in this section. We also illustrate briefly an application-specific adjective, the balanced evolution.

6.4.2.1 Spontaneous Evolutions

Some evolutions in the domains studied occur spontaneously. For example, when the requisite inputs for a batch process are available, the evolution should be automatic, i.e. spontaneous evolutions are gated only by the availability of their inputs. The semantics of a spontaneous reaction are restricted in effect to the reaction itself; they do not affect other objects or reactions, or the semantics of the object-states at a later point in time. Logically, if \(R\) is a reaction, then spontaneity is expressed by the logical formula:

\[
W = \neg\lambda x. \text{Defn(Source}_\text{Class}(R))(O)\mid_{t} \wedge \lambda x. \text{Defn(Source}_\text{Class}(R))(O)\mid_{t'} \wedge \\
\lambda x. \text{Defn(Target}_\text{Class}(R))(O)\mid_{t'}
\]

if \(I\) immediately precedes \(t\), \(I'\) immediately follows \(t\), and \(n_{\text{Source}_\text{Class}(R)}; \text{Begin(CMInt}_t) = t\).

In this case, the spontaneous reaction should take place at time \(t_1\), if \(I' = [t_1,X)\). Intuitively, as soon as \(T_0\) reaches \(n_{\text{Source}_\text{Class}(R)}\), the preconditions for the link interpreting spontaneous evolution \(R\) should be satisfied. \(W\) implies constraints of the types \(C_1\) and \(C_2\) (section 6.4.1):

1. \(C_1[II] = \neg\lambda x. \text{Defn(Source}_\text{Class}(R))(O)\mid_{t} \wedge \lambda x. \text{Defn(Source}_\text{Class}(R))(O)\mid_{t} \wedge \lambda x. \text{Defn(Target}_\text{Class}(R))(O)\mid_{t'}\)

2. \(C_2[iii] = \lambda x. \text{Defn(Target}_\text{Class}(R))(O)\mid_{t(t)}\), where \(t_1\) is the time of spontaneous evolution.

\(C_2\) is the normal post-condition on evolutions (section 6.3.3.1), hence no change is required. \(C_1\) above contradicts the default pre-conditions as given in section 6.3.3.1. We resolve the conflict by replacing \(\text{Pre}\_\text{Conds}(I(R))\) by the following:

\[
\text{Pre}\_\text{Conds}(I(R)) = \lambda x. \text{Defn(Source}_\text{Class}(R))(O)
\]

Note that this pre-condition does not require any attribute to be updated or any explicit action to make \(O\) a candidate for traversing the link \(I(R)\). Spontaneous reactions are inter-
interpreted computationally as follows (as in figure 6.10, which makes SO==> SSO spontaneous):

- If a reaction \( R \) such that \( \text{Target\_Class}(R) = C \) occurs at time \( t \), the postcondition of \( R \) establishes \( \forall x. \text{Defn}(\text{Target\_Class}(R))(O) \models t \), where \( O \) is the reacting object. This immediately satisfies the pre-condition of every link interpreting spontaneous evolutions \( R_i \) such that \( \text{Source\_Class}(R_i) = C \), because the precondition on every \( I(R_i) \) is \( \forall x. \text{Defn}(\text{Source\_Class}(R_i))(O) \). 

- Since pre-conditions are specified for the interval immediately preceding the time of reaction, and \( \forall x. \text{Defn}(\text{Source\_Class}(R_i))(O) \models t \), the reaction is possible at \( t_1 \) such that \( t \) immediately precedes \( t_1 \).

![FIGURE 6.10](image)

**FIGURE 6.10** ATN Interpretation of Spontaneous Evolutions

**Proposition 6.1** A Spontaneous reaction is modeled in d'foundation by introducing a pre-condition equivalent to a statement of its source class definition on the interpreting ATN link.

**Corollary 6.1** If no other pre-conditions apply to the interpreting ATN link \( I(R) \) of a spontaneous evolution \( R \), an equivalent interpretation is the negation of the normal pre-conditions, i.e. \( \left( \left( \text{nClass\_Candidate}(T_o) = \text{False} \right) \land \left( \sim \text{nClass}\_a(T_o) \leq \text{Val} \right) \lor \forall x. \left( \text{Defn}(\text{Class}))(O) \right) \right) \).

The proof of the corollary is obvious. Note that if more than one spontaneous evolution is sourced at a node \( n \), then the heuristics in section 6.3.3.1 may be applied to select one of the reactions. Note also that if a normal (non-spontaneous) reaction \( R \) is sourced at a node \( n \) along with a spontaneous evolution \( R' \), \( R \) will never be traversed.

### 6.4.2.2 M:N Reactions

M:N reactions require \( M \) object-states to be bound in a dynamic relationship with \( N \) subsequent object-states. As mentioned previously, MtoN is an example of an adjective which requires the analyst to provide aggregation structures (as shown in chapter 4) over which the semantics can be applied. Essentially, aggregate source and target classes \( \text{Source} \) and \( \text{Target} \) are defined. The identity of an instance of \( \text{Source} \) is made by a constructor function on the identities of all input objects. The attributes of the source (resp. target) class are typed according to the class of the corresponding input (resp. output) object-state. The input and output object-states are modeled as the values of these attributes. MtoN semantics is given by \( W \):
CHAPTER 6: D'FOUNDATION - A COMPUTATIONAL FRAMEWORK

\[ W = (\lambda x. \text{Defn}(C_i)(O_i) \land n_{\text{Source}} : r_{C_i}(T_S) = O_i \mid t, \ i = 1..m) \land \]
\[ (\lambda x. \text{Defn}(D_j)(P_j) \land n_{\text{Target}} : r_{D_j}(T_T) = P_j \mid t, \ j = 1..n) \]

where \( t \) is the time of reaction, \( C_i \)'s and \( D_j \)'s denote attributes of source and target aggregate classes, and \( O_i \)'s and \( P_j \)'s denote object-states. Abusing terminology slightly, we say that \( C_i \) (resp. \( D_j \)) also denotes the class or type of the attribute \( C_i \) (resp. \( D_j \)). \( W \) states that before \( t \), objects \( O_i \) should satisfy the type constraints of \( C_i \), where \( C_i \) is the class of the \( i \)'th input object-state modeled by the \( i \)'th attribute of the aggregate source class \( \text{Source} \) of the \( \text{MtoN} \) reaction \( R \). At \( t \), object-states \( P_j \) should satisfy the definition of classes \( D_j \), where \( D_j \) is the class of the \( j \)'th output object-state modeled by the \( j \)'th attribute of the aggregate target class \( \text{Target} \) of the \( \text{MtoN} \) reaction \( R \).

In terms of the discussion in section 6.4.1, \( W \) matches constraints \( C_1 \) and \( C_4 \) as follows:

1. \( C_1[v] = (\lambda x. \text{Defn}(C_i)(O_i) \land n_{\text{Source}} : r_{C_i}(T_S) = O_i \mid t, \ i = 1..m) \), if Condition \( i \) are unified with \( \text{Defn}(C_i) \), Relation\( (X, \text{Source\Class}(L)) \) unifies with the condition \( n_{\text{Source}} : r_{C_i}(T_S) = O_i \), i.e. \( I(X) = O_i, I(\text{Source\Class}(L)) = n_{\text{Source}} \), and \( I(\text{Relation}) = r_{C_i} \).

2. \( C_4[vii] = (\lambda x. \text{Defn}(D_j)(P_j) \land ) \mid t \land n_{\text{Target}} : r_{D_j}(T_T) = P_j \mid t, \ j = 1..n) \). This matches the Condition, Relation and related object as above, but the period of validity is the timepoint \( t \).

\( C_1 \) and \( C_4 \) are used to modify the pre-conditions, actions and post-conditions associated with \( R \) as per the stated rules:

'And'ing \( C_1 \) to Pre_Conds(\( R \)) for M:N reactions yields:

\[ (n_{\text{Source}}:\text{Candidate}(T_S) = \text{True}) \lor (n_{\text{Class}}:a(T_S) <\text{- Val} \mid t \land \neg \lambda x(\text{Defn}(\text{Source})))(O) \mid t ) \land \]
\[ (\lambda x. \text{Defn}(C_i)(O_i) \land n_{\text{Source}} : r_{C_i}(T_S) = O_i \mid t, \ i = 1..m) \], where \( \lambda x(\text{Defn}(\text{Source}))(O) \) implies \( n_{\text{Source}} : r_{C_i}(T_S) \neq \phi \) for \( i = 1..m \) (see chapter 4). This is satisfiable if the reaction is explicitly triggered by setting \( n_{\text{Source}}:\text{Candidate}(T_S) = \text{False} \) at some time \( t \). Under normal circumstances, therefore, M:N reactions can be triggered only explicitly by the user. We illustrate the combination of this interpretation with the semantics of spontaneous reactions later in this section.

'And'ing \( C_4 \) with the default post-conditions \( \lambda x(\text{Defn}(\text{Target}(R)))(O) \mid _{t,\text{now}} \) does not create an inconsistency since the default implies \( n_{\text{Target}} : r_{D_j}(T_T) \neq \phi \) for \( j = 1..n \) as well.

The source and target object classes in an M:N reaction may be considered useless in the sense that they exist only in order to capture the many-to-many semantics of M:N reactions. Consequently, M:N reactions may be designated as transformations where the source object dies. The aggregation relationships linking the (real) source objects and the "dummy" aggregate source object in an M:N reaction is a loosely bound aggregation (chapter 4). This
allows the individual objects to exist after the aggregate object dies. The target aggregate in an M:N reaction is also loosely aggregated from the N real target objects (see definition 3.9).

Proposition 6.2 M:N reactions can be modeled in d'foundation by complex source object classes aggregated through loose bindings to the M real source object-states and complex target object classes aggregated from N real target object-states through loose bindings.

(a) Modification to the Original Model

(b) ATN interpretation before adjectives

(c) ATN interpretation of <Spontaneous, MtoN>

FIGURE 6.11 Spontaneous M:N Reactions in an ATN

M:N reaction semantics and spontaneity may be combined simply by applying the rules of spontaneity to the default preconditions and then applying the M:N interpretation, to yield Pre_Conds(R) = λx.Defn(Source_Class(R))(O) ∧ (λx.Defn(Ci(Oi)) ∧ nSource : rCi(Ts) = Oi), i = 1..m), which preserves the semantics of both adjectives in an obvious way. We illustrate these concepts by remodeling the GR -/> GI part of our example in section 6.3.4 as an M:N
spontaneous reaction such that GR's produce GI's in pairs, not singly. We illustrate the differences in initial modeling, its transformation to ATN, and the changes in pre- and post-conditions introduced by the adjectives <MtoN, Spontaneous> in figure 6.11

- In (a), the existing definition is modified to create an aggregated Src object from the two GRs which are input into the reaction, and an aggregate Tgt object aggregated of the two GR objects which survive the reaction (the original GR-/> GI was a production) and the two new GI objects. The input GR objects are qualified by Survives_As(X) to indicate identity preservation in an M:N context (chapter 4).

- Relationships among source and target object-states' attribute values are specified in a way similar to 1:1 reactions; we do not elaborate further.

- The * in figure 6.11 demonstrates an additional (optional) action while modeling spontaneous M to N reactions: The action of assigning a newly created GR instance to a Src object which is waiting for inputs for an M:N reaction. In this case, the T_{GR} is assigned to PublicSrc.i(next), where PublicSrc is a global register pointing to the next waiting instance of the class Src, and next is another global register indicating the attribute index on the object where T_{GR} may be assigned. The user may manually assign T_{GR}'s to waiting Src objects. This depends on the desired interpretation of spontaneity.

- (b) shows the interpretation of the structure and actions in the context of the post-conditions implied by (a), i.e. (\lambda x.Defn(Dj)(Pj) \land \land \land n_{\text{Target}} : r_{Dj}(T_{\text{Tgt}}) = Pj) \land j = 1..n). The structure of the interpretation has been explained. We examine the actions in detail:
  - The second conjunct of the post-condition above leads to the generation of actions of the form labeled by A and B in the figure:
    - A assigns T_{GR1} and T_{GR2} to the appropriate registers as indicated by Survives_As(o1) and Survives_As(o4) respectively, ensuring preservation.
    - B attempts to create new identities for registers o2 and o3 in order to satisfy the definition of Tgt. In the absence of any design-time specification of how the identities are to be created, the user must specify at runtime if \text{Id}(T_{GII}) = f(\text{Id}(T_{GR1})) or \text{Id}(T_{GII}) = f(\text{Id}(T_{GR2})), and whether f is a constructor or evaluator function. This can be optionally specified at design time.
    - The first conjunct of the post-condition leads to the generation of actions of the type C (satisfying definitional conditions of T_{GR1} and T_{GR2}), and D (satisfying definitional conditions of T_{GII} and T_{GII}) by assigning appropriate values to the different registers (Item_Name, Item_List and Req_Date) of types A through I as given in section 6.3.3.7, from previous values or user input at runtime. This process has been illustrated earlier, and is omitted from this example for brevity.
    - These actions together satisfy the actual and dummy class definitions. Note that when Source and G-Source of the target object-states are referenced, the object T_{Src} will be indi-
cated, since there is no direct link between the actual source and target object-states other than through $T_{src}$ and $T_{tgt}$.

- (c) illustrates the pre-conditions as modified by the interpretation of the adjectives $<MtoN, Spontaneous>$ making this reaction an M:N and spontaneous one, as previously explained in this section.

### 6.4.2.3 Reversible Reactions

Reversibility is a common guarantee for certain steps of a process. If reversed, a reaction must appear as though it never happened, which implies a guarantee of the ability to recreate the source object in the future (unless the user explicitly allows an irreversible reaction involving the target object state or its descendants). Formally, we may express this requirement logically through $W$ as follows:

$$W = \bigcup Object\_State(P_j) \mid t' \vdash Object\_State(O) \mid t$$

where $O$ undergoes a reversible reaction $S >> T$ at time $t$, $t'$ immediately precedes $t$, $t'$ is a timepoint = $t$ or after $t$, and $P_j$ is an object that exists at $t'$, and $P_j = O$, or $P_j$ is generated directly from $O$ by a reaction that occurs at or after $t$, or $P_j$ is generated directly from another object $O'$ which is directly or indirectly generated from $O$ at or after $t$.

Computationally, this imposes the meta-condition that the identity and all attributes of the source object-state in a reversible reaction should remain derivable indefinitely into the future. This condition may be violated if an attribute of a direct or indirect target object-state is derived through an evaluator function that consumes the source object attribute, or if the identity of an object generated from a target object-state is computed using an evaluator function in the future. Consider a reversible reaction from a source class having one attribute $A$ to a target class having one attribute $B$, such that $B = f(A)$. The reversibility of this reaction can be assured if $f$ is reversible, i.e. is a constructor. (Constructor and evaluator functions, and their use in generating identities, is discussed in sections 4.2.1 and 4.3.1). Note however that attribute and identity relationships can be expressed through constructor functions only if the domain supports such a definition. In some cases this is not possible, such as in the case of a nuclear fission transformation that cannot be reversed; in other cases, it is possible, as in the case of transforming pulp to paper. The important consideration is to model truly reversible identity and attribute relationships through evaluators. $W$ implies the following:

- $C_4[\forall] = \forall n_5: r_{aj}(T_0) = v_{aj} \mid t \quad (\land_1 n_T: r_{bi}(T_T) = v_{bi} \mid t \vdash n_5: r_{aj}(T_0) = v_{aj} \mid t'$), where $\vdash$ is the derivability relationship. $C_4$ says that the set of register values at $T$ should be sufficient to derive every register value at $S$ before $t$. This follows directly from
the definition of object-state and W. This implies two constraints:

- If $T_T \neq T_O$ and $Id(T_T) = Create\_Identity(Id(T_O))$, then $Create\_Identity^{-1}(Id(T_T))$ exists, or equivalently the meta-condition $Constructor(\text{Create\_Identity})$.

- If $(n_T:r_b(T_T) = f(\ n_S:r_a(T_O) = v_{a_j} |_{t'} ) = v_{b_j} |_{t})$, then $f^{-1}(v_{b_j}) |_{t'}$ exists. This follows directly from the definition of object-state as a conjunction of attribute values and W above. Equivalently, $Constructor(\text{Create\_Identity})$.

- $C_4[vii] = \forall n_S;r_a(T_O) = v_{a_j} |_{t'} (\forall X (A,n_X;r_b(T_X) = v_{b_i} |_{t''} \land Derived\_From(X,O)) \Rightarrow n_S;r_a(T_O) = v_{a_j} |_{t'})$, where $\Rightarrow$ is the derivability relationship. This follows directly from the definition of object-state and W. It states that if $T_X$ is a token denoting object $X$ that is related to $O$ by the Derived\_From relationship, then the conjunction of statements regarding attribute values of $X$ at $t'$ should be sufficient to derive every register value of $O$ at node $S$ during $t'$. The constraint matches $C_4[vii]$ if we unify Relation with Derived\_From, with respect to the source class $S$. As above, this implies two constraint types:

- We define Derived\_From$(X,O)$ as follows: $T_X = T_O$ or $Id(T_X) = Indirect\_Id(Id(T_o))$, where

  $Indirect\_Id(Id(T_o)) \Leftrightarrow Create\_Identity(Id(T_o))$, or

  $Indirect\_Id(Id(T_o)) \Leftrightarrow (Create\_Identity(Id(T_o)) \land Id(T_y) = Indirect\_Id(Id(T_o)))$

  In order that $T_o$ is derivable from $T_X$, it is necessary that for all $y$ such that $Id(y) = Indirect\_Id(Id(T_o))$, $Indirect\_Id^{-1}$ should exist, i.e. $Create\_Identity^{-1}(Id(T_Y))$ exists, or $Constructor(\text{Create\_Identity})$, in every intermediate generation.

- Similarly, we require $Constructor(f)$ for every $f$ such that $n_X;r_b(T_X) = f(\ n_Y;r_a(T_Y) = v_{a_j} |_{t'}) = v_{b_i} |_{t''}$, where $X$ and/or $Y$ are Derived\_From $O$ and $t''$ and $t'''$ are time points after $t$. We omit the reasoning since it follows from the above for identities and $C_4[vii]$.

In our framework, the special register Save\_Set (definition 6.2) is used to upload, propagate and download registers on which constraints apply in the future with respect to a reaction. As per the guidelines in section 6.4.1, we modify the ATN constructs as follows:
• Post-Conds(R) <- Post-Conds(R) ∪ Constructor(f) ∀f such that ∃nT:ri(T) = f{|nS:raj(T) = vaj|P}. If the reaction is a generation, then Post-Conds(R) <- Post-Conds(R) ∪ Constructor(f) ∪ Constructor(Create_Identity). These conditions ensure that all immediate actions in the reversible reaction are in fact reversible.

• To ensure reversibility in the future, we include the following actions: Save_Set(R) <- Save_Set(R) ∪ Rev_Constraints. Rev_Constraints is the set Id(O) ∪ ∀j nS:raj(T). ∀X ∈ Rev_Constraints, Condition(X) is defined as ∀Y∀R (nY:rR = f(Z) ∧ X = Z) => Constructor(f). The operator ε* is defined as follows:
  • Xε* Z = True if y ∈ Z, y is a simple value and X = y, or
  • Xε* Z = True if y ∈ Z, y = f(W) and X = y, or
  • Xε* Z = True if y ∈ Z, y = f(W) and Xε* W.
  • Otherwise, Xε* Z = False.

Informally, if either the Id(O) or any of its attributes are used to generate any other identity or attribute value, the generating function must be a constructor, so that it can be reversed.

We illustrate the interpretation of reversibility by making the MO -->GR transformation reversible in figure 6.12. Initially, Save_Set is empty. When the reversible MO --> GR occurs, it is initialized with the identity of the MO instance, i.e. O and the only attribute of MO's, i.e. Matl_Date. At the same time, the post-conditions check that the identity creation function f that creates the identity of the GR, i.e. f(O) is a constructor, so that O can be retrieved from f(O). It also checks that the derivation of the GR:Req_Date attribute from the MO:Matl_Date attribute is a constructor, so that the Matl_Date is not lost (it is, since GR:Req_Date = MO:Matl_Date). The later reaction GR=/>= GI also checks to ensure the derivability of the members of Save_Set. In our example, it only has to check to make sure that GI.Req_Date = GR.Req_Date is a constructor (it is), because GR.Req_Date is derived from MO:Matl_Date, which must remain derivable. To conclude, we state the proposition we have argued in this section below.

**Proposition 6.3** Reversible reactions can be modeled in our framework by associating
the derivability post-condition with the identity and every register of the source class, and including it in the distinguished register Save_Set.

**6.4.2.4 Balanced Reaction Example**

If the semantics of any arbitrary adjective can be described logically and analyzed as in our examples above, then it would be possible to interpret in terms of an ATN-based com-
putational framework within the limits already stated. As example, the requirements of the maintenance management domain include the need to model balanced evolutions. Intuitively, a balanced evolution seeks to ensure that the object does not traverse a given state more than a certain number of times in its future evolution. For example, after a certain number of times (parameterized by "n"), a circuit card will not be allowed to enter a fault state. Logically, we express this as:

\[ W = n_X:Bal\_Count(T)=n \mid t \Rightarrow n_Y:Bal\_Count(T)=n+1 \mid t \land n_Y:Bal\_Count(T) < n \mid t, \]

where the qualified reaction is \( X \gg Y \), the reaction occurs at \( t \), and \( I \) immediately precedes \( t \).

The first conjunct of this constraint matches our template \( C_3[v] \), leading us to modify the post-conditions on the links interpreting \( X \gg Y \) as follows:

\[ \text{Post\_Conds}(L) \leftarrow \text{Post\_Conds}(L) \cup (n_X:Bal\_Count(T)=n \lor n_Y:Bal\_Count(T)=n+1) \]

This in turn leads to \( \text{Actions}(L) \leftarrow \text{Actions}(L) \cup n_Y:Bal\_Count(T) \leftarrow n_X:Bal\_Count(T)+1. \)

The second conjunct of \( W \) matches our constraint template \( C_2[iii] \) and can be 'and'ed to the post-conditions with no impact on existing constraints. In order for this semantics to succeed, the user-defined register \( Bal\_Count(T) \) must be initialized to 0 and attached to every node \( X \) and \( Y \) such that the reaction \( X \gg Y \) is qualified as balanced. It is easy to see that this condition preserves the desired semantics of balanced evolutions, and this case study concludes our argument for the usability of ATN interpretations to enforce adjective semantics.

### 6.5 Reasoning with Grammatical Descriptions

Sections 6.3 and 6.4 introduced the interpretation of d'dialect specification of dynamic relationships in terms of the computational mechanism of ATNs. This interpretation allows us to adapt formal language algorithms to reason with dynamic relationships of individual objects, as well as with dynamic behavior of schema. Specifically, we can:

- Enforce checks, actions and integrity constraints associated with legal changes in the objects in a database, to ensure data consistency in a changing environment.
- Trigger reactions based on incremental updates, based upon the knowledge of the global processes of change in the domain. A series of reactions may be triggered by a single attribute update, since the grammatical definition of processes of change enable us to look ahead to rationalize the attribute update in terms of the eventual state of affairs derived from the grammatical definition.
- Infer historical states of affairs, and allow future hypothetical and modal queries.
• Control the cost of maintenance of dynamic constraints, based on an understanding of the cost of parsing sentences derived by grammars of different expressive powers, augmented by conditions and actions, each with its own cost.

• Modify the description of dynamic behavior of the schema or object base, ensuring that the changes preserve domain-inherent constraints such as unambiguity.

• Group changes to a set of objects as a transaction, and enable backtracking over them as a unit, to avoid inconsistencies from a global conceptual perspective.

Though the above uses of the grammatical properties of d'foundation address the concerns of objects, they are extensible to changes in schema as well (We do not develop this topic in detail in this thesis. Please refer to chapter 8). For schema, the advantages of a formal representation of allowable changes is important for the following reasons:

• Changes in schema can be triggered to optimize performance or expressibility, based on changes in the meta-properties of the schema.

• Schemas can be integrated based on a formally defined process of integration for generic schemas of that type. Integration may or may not be information-preserving, depending upon the requirements on the information system.

• A grammatically defined process of change can be used to incrementally evolve the schema, including migration of the instances, on a lazy, when needed basis.

• Allowable schema changes can be pre-specified, in order to control the churn in distributed schemas and data organizations which are common today.

Comparison with Petri Nets shows the advantages of a grammar-based formalism over those based on communicating processes for dynamic and adaptive information systems.

• Dynamic and adaptive information systems emphasize object-centric incremental attribute-oriented change over modeling of concurrent processes. Petri Nets lack devices to associate and analyze substantial information attached to the tokens (objects) that transit through the process. As well, individual steps of Petri Net processes are opaque, and can not be embellished with actions and conditions.

• Petri Net analyses are based on macroscopic properties observable at its boundaries, such as reachability and races (supported, but less efficiently, by ATNs). Semantic equivalence of process goals is not addressed by Petri Nets.

• The important aspect of process parallelism for information modeling is transactional synchronization (the notion that two stages must be co-ordinated, or must together precede, a third step). It is not crucial to the informational model that the steps actually occur in parallel - that is a performance issue. Thus, one of the advantages of Petri Nets as a tool is of secondary importance in modeling.
Whereas ATNs describe and parse text ordered in space, dynamic and adaptive information systems demand facilities to parse and backtrack in time to deduce and update attribute values of sets of objects. This provides the ability to deduce missing information, reason about necessities and possibilities, and deduce past and future states of affairs, actual and alternative, from information at a given time. Petri nets lack such capabilities.

The ATN-based algorithms of interest are those which promote the kinds of deduction and analysis found in formal language theory. These algorithms carry a cost of computation ranging from logarithmic in terms of the length of the inputs, to exponential. In some cases, computations can be undecidable (as in grammars defining certain recursively enumerable sets [79]). However, restrictions imposed on the conditions, actions and structure of the ATN descriptions are instrumental in limiting the penalty to acceptable levels.

In section 6.7, we introduce two types of reasoning in the object domain. We introduce some basic algorithms which address these needs and refer to others outside the scope of this thesis which may be modified for yet other operations in section 6.8. We also discuss the complexity of operations using these algorithms given the data structures of chapter 7.

6.6 Expressive Power of Histories

The usability of the grammar-based ATN formalism comes from the analogy of sequences of object-states in time to sequences of symbols in space. Similarly, questions regarding the generation of strings by alternative parse trees have directly analogous questions in the domain of generation of alternative plans in dynamic information systems.

Formal grammars are the basis for generating, manipulating, and reasoning with the former, and d'foundation applies the well-understood power and expressibility results from formal language to the latter. The key to this application is the interpretation of object-state sequences as strings of symbols. A temporal sequence of object-states is implicit in the stored record of an object's attribute values over a period of time, given intensional class definitions. We use the term recorded history (or history) to refer to this sequence of object states.

**Definition 6.5** An *recorded history* for an object id is the record of its object states over time. This record is implied by the intensional description of classes as applied to the union of attribute value sets of the form < id, a, \{ < v, [t1, t2] > \}, for every a that is an attribute of id.

Specific dynamic features do affect the complexity of computations, but the most basic source of complexity in such a grammatical characterization is the structure of histories
allowed by the system. The kinds of histories allowed also determine the expressiveness of models, implying a trade-off between expressiveness and computational complexity.

At one extreme, conventional databases store only the current state of affairs. In grammatical terms (oversimplified), each object in the database, and each attribute of every object, is governed by rules of the form:

\[ A \rightarrow B \]

where \( A \) and \( B \) are both non-terminal symbols in the grammar. The database remembers nothing in the past, generates no string (except the degenerate unitary string consisting of one symbol) and in our terms, generates no history.

At the other end of the verbosity scale, conventional temporal databases store every event and state of every object and attribute for all time, quite inefficiently. If temporal databases were meant to capture and be governed by rules regarding valid changes (which they are not), the rules would be characterizable through right linear (regular) rules of the form:

\[ A \rightarrow \alpha B \]

Where \( \alpha \) is a string of terminal symbols, \( A \) and \( B \) are non-terminal symbols which can be further expanded (lead to further evolution of states through time under our interpretation). Although temporal databases are the most verbose, they can not represent semantically complex historical patterns of object-state changes through time. In practice, frequent updates in temporal databases generate histories of large sizes. The sizes of the histories generated make all but a few types of queries (such as finding specific attribute values for entities of a fixed class at a specific time) inefficient. Frequently, the primary use of temporal data derives from the ability to archive data in the sequence of its creation.

ATN-based descriptions allow reasoning with information over and above executing queries with respect to entities and attributes with an additional temporal index. They allows us to define and reason with histories which are

- succinct - information can be summarized in terms of successive significant snapshots, and historical object states can be replaced by other object states, or omitted from the database if, as per the dynamic domain definition,
  - they do not need to be stored explicitly; and/or
  - the information can be derived from historical and future states if required
- expressive - an object state can constrain future object states.

In the context of storing and reasoning with changing representations of an object’s
migration through different states over time, we define derivation histories as follows, and as illustrated in figure . (For definitions of the terminology, please refer to [79].)

**Definition 6.6** A *derivation history* associated with an recorded history $H$ of an object $O$ is the directed graph defined by successive applications of production rules from a grammar, such that the resulting sentential form comprising a record of memberships of $O$ in various classes over time is $H$.

**FIGURE 6.13** Recorded and Derivation Histories

If the grammar describing an object's dynamic behavior is ambiguous, there may be more than one derivation history associated with a recorded history. The recorded and derivation histories associated with an object are used in temporal inference regarding the object's attributes, as well as modal queries. It is natural in our opinion, to characterize databases in grammatical terms (which is admittedly self-serving, since ATNs are grammar-based). We propose some points of interest with respect to expressiveness of dynamic databases, and associate them with the power of the grammar required to model them.

### 6.6.1 Characterization of Historical Databases

Practically useful descriptions of dynamic behavior corresponding to each of the major grammatical categories exist. We motivate the need for algorithms that can reason with all of these types of histories through some examples of real-life histories. There is no independent justification for classifying recorded histories into these categories, apart from their correlation with grammatical ones. The definitions below are therefore in the nature of motivating examples; they do not contribute significantly to our results.

In this section, we use capitals to denote "current" states and small letters for past or future states. Conventional (non-temporal) databases which store no information about past states have degenerate (null) histories where all historical information is deleted.

**Definition 6.7** A *degenerate* object history is one where the history prior to the distinguished point in time $t_{\text{now}}$ is null (contains 0 object states).
Regular grammars describe histories where no information is deleted, and every state is dependent only on the current state immediately before the state-creating event.

**Definition 6.8** A **situated** object history is one where dynamic relationships established at time t depend only on the object-states immediately before t, and no historical information is deleted.

**Example 6.1** A graduate student migration schema expressed by the following rules given a situated history

Applicant => applicant.Student
Student => student.Alumnus

Conventional historical databases can be specified by regular grammars, such as:

A -> aA, A -> bA ... for every a, b ... belonging to the set of object states.

Besides leading to a huge description, this method of specification does not capture any constraints on valid evolution patterns. Situated histories can be concise and yet capture the dynamic domain knowledge used to guide subsequent dynamic relationships. Context free grammars can form the basis for expanded object histories.

**Definition 6.9** An **expandable** object history is one where the record of an object o's membership in a class C during \([t_1, t_2]\) may be further elaborated by the record of its membership in a sequence of classes C during \([t_1, t_2]\), at a later point in time.

**Example 6.2** A grammar for the (expandable) history of a registered parcel including notification of receipt may be expressed in part by rules such as the following:

In_transit_parcel =>
in_forwarding_parcel.In_transit_parcel.receipt_acknowledged
In_transit_parcel => Received_parcel.

Expandable histories are found in planning applications that proceed iteratively in defining plans at greater levels of detail depending on dynamic circumstances. Resources may be committed at a high level far into the future, but the low-level details regarding deployment are recorded later, replacing the commitment information which becomes redundant as the details become available. In terms of the length of the string of object-states in time, expanded histories are length increasing.

**Example 6.3** When a new aircraft is purchased by an airline, its lifecycle plans may be described by the following grammar:
Primary_transport. backup. cannibalize

Later, the Primary_transport role may be refined to read as follows:

VIP_Transport.Transpacific.Other.backup.cannibalize

Later yet, the cannibalize stage may be refined as follows:

VIP.Transpacific.Overland.backup.engine_source.fuselage_source

The statements above are meant only to motivate the need to handle these different types of histories in the practical world.

### 6.6.2 More Complex Practically Motivated Histories

There are also other practically useful histories which may be used to make aging information more succinct. In this category are included histories whose length may be allowed to decrease locally and monotonically as per a dynamic definition. In the interest of computability, unrestricted grammars (which are not parsable) can not define histories.

Context-sensitive grammars motivate context-constrained histories as follows.

**Definition 6.10** A context-constrained object history is one where a subsequence \( C \) of the recorded history may be replaced by a different subsequence of classes \( C' \) of equal length or more (in terms of membership in classes).

Migration schemas which specify conditions such as, “Candidates must hold an equal number of training, operational and administrative posts before reaching class \( Y \)” imply context-constrained histories. This is because a local change to a class in the recorded history may create new obligations in terms of memberships in other classes at other (non-local) positions in the recorded history.

In certain instances, it may be useful to reduce the length of the stored history in a controlled way as a form of archival for aging information. This requires a restricted form of a type 0 grammar, which is used to model what we refer to as a forgetful history.

**Definition 6.11** A forgetful history is one where subsequences in the history may be replaced by other subsequences with strictly less information in terms of length of the subsequence, forever in the future.

**Example 6.4** If a database scheme for graduate students deletes student records when alumnus status is reached, a forgetful scheme may be employed as follows:

\[
\text{applicant.msc_student.Msc_alumnus} \Rightarrow \text{reduced_msc.Msc_alumnus}
\]
Forgetful histories are useful as regimes to retire (forget) dated information. In general, length-reducing productions imply unrestricted grammars whose sentences are not computable. However, under the following restrictions, decidable algorithms exist for such histories.

- The grammar is a tuple $G = \langle S, I, D, P_I, P_D \rangle$, where $S$ is the starting symbol,
- $I$ is a set of grammar symbols labeled length-increasing symbols such that $S \in I$,
- $D$ is a set of length-decreasing symbols such that $I \cap D = \emptyset$,
- $P_I$ is the set of length-increasing productions of the form $A \rightarrow B$, where $|A| = 1$ and $B \geq 1$, and $A = I^+$, and $B = I^+$,
- $P_D$ is the set of length decreasing productions $X \rightarrow Y$ such that $|X| > 1$, $|Y| = 1$, and $X = (I \cup D)^+$, and $Y \in D$. To further simplify our algorithms for parsing sentences generated by such grammars, we stipulate that if $X_1 \rightarrow Y_1$ and $X_2 \rightarrow Y_2$ are two productions in $P_D$, $Y_1 \neq Y_2$.

If sentential forms locally either increase monotonically or decrease monotonically in length, sentences generated by such a grammar can be parsed. As grammars, this class may not be strictly context-sensitive or unrestricted; equivalent CFGs may be found to describe the language. An equivalent context-free grammar, however, may not capture the visible change of the recorded history over time as the equivalent context-constrained or forgetful grammar, making it unusable in dynamic and adaptive information systems. We do not distinguish terminal from non-terminal symbols as explained in section 6.1.1, without cost penalty. Appendix E shows how length reduction and string replacement can be specified through ATNs.

Example 6.5  The two grammatical descriptions (1) and (2) below describe the same sentence "xy" if only the sentences comprising terminal symbols are considered, however, they are different in terms of the intermediate sentential forms that they generate.

(1)  $S \rightarrow xy$ ; a normal grammar
(2)  $S \rightarrow ABC$ ; $AB \rightarrow x$ ; $C \rightarrow y$ ; a forgetful grammar

In dynamic descriptions, where non-terminal and terminal symbols are not distinguishable, and intermediate sentences are valid recorded histories, the grammars are distinguishable.

More complex histories may exist [155]. There are recursively enumerable histories (migration inventories) which are not recursive. As practical devices, however, they do not interest us in this thesis, since computability problems accompany their use. The interesting histories lie between strictly regular and recursively enumerable sets.
6.7 Operations on Dynamic Objects

Queries and commands on dynamic objects and schemas can be expressed in grammatical terms because d’dialect is interpreted by a grammar-based computational model. We propose a skeleton of a simple language for modal and material queries and material updates which may be used to express such operations, and use the constructs of this language to introduce the core algorithms for d’foundation in section 6.8.

6.7.1 Operations in a Dynamic Data Manipulation Language (DML)

Dynamic data manipulation languages can be defined by enhancing the syntax of existing query languages. DML itself is not a major contribution of this thesis, and we do not specify the complete syntax (in practice, graphical languages may be preferred as a human machine interface). We emphasize, however, that the execution semantics of DML operations can require the use of algorithms for parsing grammatical descriptions, even if the syntactic expression of the operation is similar to that of a non-dynamic information system. Operations on dynamic information systems can be classified roughly into three categories:

- Queries addressed to the schema, as exemplified in section 6.7.1.1. Since these refer to well-known analytical methods from formal-language theory, we do not address them in this thesis. However, we do claim the executability of such queries as a major advantage of our method of dynamic modeling, missing from all other methods.

- Operations on present, past and future states of dynamically defined objects, in sections 6.7.1.2 and 6.7.1.3. The algorithms supporting these operations are the core operations discussed in section 6.8.

- Operations that build upon the previous category by imposing goal-directed search and modal searches on the results, as exemplified in sections 6.7.1.5 and 6.7.1.4. We do not discuss these in detail in this thesis, but outline them in order to illustrate further benefits of modeling under our proposed formalism.

6.7.1.1 Meta-Level Queries

Certain questions may address the systemic properties of specific dynamic schema, in terms of analytical meta-queries about the grammar, without reference to any specific object(s) or classes. Examples of this category are:

- Is it possible for any process (history) to continue forever?
- Is it possible to generate the same (particular) state sequence (recorded history) through two different processes (derivation histories)?
CHAPTER 6: D'FOUNDATION - A COMPUTATIONAL FRAMEWORK

In linguistic terms, these translate to:

- "Can the grammatical representation generate sentences of infinite length?"
- "Can there be two derivation trees for the same sentence?"

These questions address the meta-properties of the grammatical definition, such as ambiguity, ability to generate infinite-length sentences (finiteness of the language). They may be expressed as satisfiability of logical formulae on variables qualified universally or existentially over recorded or derivation histories. For example:

- \( \exists X \left( \forall t \left( Y \in \text{Collections}(X) | t \Rightarrow \exists t' \left( Y' \in \text{Collections}(X) | t', t' > t, Y =/= Y' \right) \right) \right) \): Is there an entity such that for every object-state, it has a subsequent distinct object state? In other words, are infinite length histories defined?

- \( | \{ \text{History}(X) \} | = \infty \), if \{History(X)\} is the set of all temporal evolution sequences that can be generated starting from the state X: Is the number of histories generated by the dynamic description starting from a state X bounded? Analogously, is the language defined by the grammar infinite?

- \( \forall H, H' \left[ ( H \equiv H' ) \Leftrightarrow ( \text{Derivation_History}(H) \equiv \text{Derivation_History}(H') ) \right] \): Is it the case that isomorphic histories have isomorphic derivation graphs? Analogously, is the grammatical description of the domain dynamics ambiguous, and is the language itself inherently ambiguous?

Though posed as logical queries, search is not a good execution mechanism for these questions. Meta-level queries on grammars are analytical, and map onto well-known analytical results and methods from formal language theory, as suggested alongside the examples above. The utility of these queries is obvious; for example, the meta-level query concerning ambiguity of the grammatical description questions regarding whether alternative plans for a given end state or a given history can be generated. We omit a review of ambiguity and infiniteness from formal language theory, and refer the reader to [79]. Note, however, that these queries are possible because of our grammar-based foundation, in contrast to others.

6.7.1.2 Material Queries

Material queries posed with respect to a domain with specified dynamic behavior address specific attributes of specific objects. Depending on the addressed attribute, queries can involve deductive reasoning with propositions at particular temporal indices, standard database operations, and/or temporal reasoning based on parsing/generation algorithms operating on the object's recorded history or derivation history, as appropriate. In terms of d'dialect, when a dormant attribute is evaluated or a temporally-determined attribute que-
ried, the grammar may be used as a recognizer (parser) to retrieve the value from the object’s previous membership in A or another class in its derivation history. For example:

- "What is the grade of service provided by a transmission line B, as it is restored to IN_SERVICE state from being OUT_OF_SERVICE?". The domain semantics may specify that the grade of service is derived from a previous IN_SERVICE state or a COMMISSIONED state on the object’s derivation history.

- If personnel are assigned to tours of duty alternating between <foreign> and <domestic>, each of which can be further qualified by a set of at most 5 (perhaps concurrent) <administrative>, <fieldwork> and <teaching> assignments, then the following query requires grammatical reasoning along the dynamic dimension:
  "How much time may a person A spend abroad before repatriation (Given her history to date)?"

- "Which of the current managers attained the highest possible academic qualifications in their career paths achievable by people with a foreign degree?"

Queries such as the above may require inference of attribute values from states in the recorded historical (or future) states based on attribute definitions. If the information is not in the recorded history, it may be parsed to get the derivation history, which may allow the derivation of the result. Material queries are posed by existentially quantified variables in temporal or historical propositions. Type (1) and (2) selection queries below are examples.

**Queries on Temporal Propositions**

(1) SELECT <Attributes> FROM <Set> ;
WHERE <Conditions> ;
{ AT | DURING | ((IMMEDIATELY) BEFORE | AFTER )
<Interval> | <Temporal_expression> } ;
{ IMMEDIATELY (PRECEDING | FOLLOWING) <Class> ) |
( G_SOURCE <Class> ) } ;

The example illustrates a temporal-SQL-like syntax. Our extension to the syntax concerns the specification of a point in the lifecycle of an object in terms of the reactions that it may undergo, or in terms of the source or generative source of the queried object-states.

**Queries on Historical Propositions**

(2) SELECT <Attributes>
FROM <Set> ;
WHERE <Conditions> ;
{ AT | DURING | ((IMMEDIATELY) BEFORE | AFTER )
<Interval> | <Temporal_expression> } ;
ON_HISTORIES <Historical_pattern> ;

The syntactic enhancements to a temporal-SQL-like language are again small. The "ON_HISTORIES" clause restricts the context in terms of the recorded history of the objects. Supporting definitions include the following:

- \(<Temporal_expression>\)s may be of the form \(\text{WHEN } <\text{Condition(Object)}>\)
  which evaluate to an interval or set of intervals during which the conditions specified with respect to a set of attributes are satisfied.

- \(<Conditions>\) refer to relational/object (O/SQL) clauses on complex entities, e.g. \(\text{father->birthdate(X)} < 1/1/52\)

- \(<Historical_pattern>\)s are currently restricted to linear subgraphs denoting a pattern of object-states in the recorded history of an object. Partially specified and branching patterns will be studied in the future.

- \([\text{status}_\text{busy}, \text{standby}, \text{in-service}]\) signifies a subgraph of object states starting with the state of an element being "busy" and ending with it being "in_service", with some states in the interim, as shown in figure 6.14.

FIGURE 6.14 Example of a Linear History in a Query Domain Constraint

We discuss the use of the historical patterns as a query constraint peripherally in section 6.8, because it is a special addition to the core algorithms, as in the case of operations in sections 6.7.1.2 and 6.7.1.3.

### 6.7.1.3 Material Actions

A grammar-based formalism can reconcile and validate incremental attribute updates with the global dynamic semantics. When an incremental update indicates the establishment of a dynamic relationship between object-states, it can be triggered according to the rules in section 6.3. Operations that may trigger a reaction include the following:

- If an update causes an object to cease being a member of an existing class, legally related object-states which it may be in the process of establishing a dynamic relationship with should become candidates as reaction targets. For extensions and productions, the update need not negate a current class membership.

- A user can explicitly effect a reaction for an object.

- If a user explicitly requests that an object be placed in a specific class, the system must ensure that the requested membership is valid in the current context, given the dynamic domain rules, either through a direct dynamic relationship between
the object-states, or through a sequence of successive reactions (history)

Under all circumstances, the appropriate associated actions on the concerned objects must be effected and the integrity constraints preserved. The operations involved are:

- **Type (3) update operations**
  
  (3) UPDATE <Object.Attribute> TO <Value> AT <Time_point>

  Object.Attribute can be an object-oriented attribute. The syntax of this operation is very similar to other languages. The difference lies in the semantics of an update, which may trigger a reaction in <Object> or other related objects in a dynamic information system. If more than one reaction might be implied because the update is non-specific with respect to the target class definitions of the reactions incident on the current class, the system must support heuristics of choice for the user. Examples of such updates include:

  - When the CGPA of a student falls below 3.0, a reaction removing the student from the program may be triggered, with or without user intervention.

  - In a chemical data acquisition database driven by periodic concentration counts, a measurement showing a rise in water concentration may need to invoke disambiguating rules to determine which of the following reactions should be recorded:

    (a) ZnO + 2HCl -> ZnCl₂ + H₂O ; or

    (b) ZnO + H₂SO₄ -> ZnSO₄ + H₂O

    in the context of measurements showing ZnO in the reaction vessel.

- **Type (4) class-related update operations**

  (4) CHANGE-CLASS <Object-State> TO <Class> AT <Time_point>

  The operation above explicitly requests a change in the categorization of an existing object, which can imply intermediate object-states involving other classes. The process of making the required changes may in turn involve further queries and updates. This involves the use of the grammar-based definition to decide how to evaluate the attributes of the new and existing class to satisfy the semantics of the reaction undergone. For example:

  - If a reaction makes an alumnus of the masters program an applicant into the Ph.D. program, some facts related to the previous graduation (e.g. the previous student id) may need to be preserved, others such as birth-date may be brought forward to the new instance of student-ship, and yet others such as last-semester-as-student may need to be replaced.

- **Type (5) reaction-related triggering operations**

  (5) TRIGGER <Reaction> ON <Object-State> AT <Time_point>
A change may be explicitly requested in an equivalent manner by demanding that a specific reaction be carried out at a given object-state. As in the case of changing class of object(s), triggering reactions can lead to further reactions or queries on related objects.

- When a user changes locations, her telecommunications service profile for follow-me services must be moved from one database to another geographically separated database, perhaps gaining or losing certain attributes and changing format, as per the restrictions of the source and destination databases.

- Type (6) Reversals of material actions.

(6) **REVERSE**<Object-State> { TO(<Class>|<Time_point>) | VIA <Reaction>}

An existing dynamic relationship may be annulled (as though it never happened) through the reversal request. Reversal may be requested with respect to a class in the relative past, or with respect to a time point in the relative past. Reversal may specify a reaction via which the reversal should take place. Reversal is allowed only if all intermediate reactions were reversible, and it may also have side effects on other objects, including the destruction of some (if a generation is reversed) objects created in the past. The semantics of this request may require user-mediation in order to achieve the intended semantics of the operation.

### 6.7.1.4 Goal-Directed Updates

(7) **TRANSFER** <Object-State> TO <State> { AT | DURING | (IMMEDIATELY) BEFORE | AFTER ) <Time_point> }

In some real-world scenarios, the core generation and parsing algorithms are useful when combined through goal-directed evaluation schemes. **TRANSFER** is motivated by planning-type applications where grammatical methods are applied in reaching goal states in the past or future which conform to specific conditions specified on a set of objects.

**FIGURE 6.15** Example of Goal-Directed Operations

- If the target state is in the past, but not reversible from the current state, TRANSFER may generate history to a state from which the past state is reachable based on a different derivation.
If the target state is in the future but not reachable from the current state, goal-directed TRANSFER may REVERSE to a state from which the target is reachable, and then generate a new history, as in the following example.

For example, the following operation: "Produce MIX from SCRAP object O" may not be doable if reaction A is irreversible (figure 6.15). However, if link C is traversed, D may be reversed later to reach the desired MIX state (assuming D is reversible) In grammatical terms, this command would require generation of recorded history to a sentential form from which a different sentential can be grammatically backtracked over.

6.7.1.5 Modal Queries

An important set of extended queries posed against specific objects in planning and design scenarios are of an exploratory (hypothetical) nature with respect to the grammar-based definition of dynamic behavior for the domain. In a fault management scenario in a large computing or communications network, examples of such queries are:

"If card A is not immediately taken offline, is it necessary that service B can not be provided at some time in the future?" and "Is it possible that if reaction X is effected, there can be a future state where quantity Q is increased?"

We label these queries modal because they involve reasoning with modal operators such as necessity/possibility on material propositions. In this case, the ATN dynamic rules are interpreted as expressing reachability relationships among snapshots of object-states.

Modal queries are posed as true/false questions against conjunctions or disjunctions of propositions with modal operators. The propositions may address temporally indexed domain objects (Temporal Propositions) or partial histories (Historical Propositions), as used in type (1) and (2) queries. The general form of modal queries is:

\[
(8) \text{NECESSARY} ( \text{Proposition}, n ) \\
\text{POSSIBLE} ( \text{Proposition}, n )
\]

where Proposition ::= Temporal Proposition | Historical Proposition, the query is implicitly posed at t_{now}, and n bounds the length of the history in order to make the process tractable.

Modal queries essentially aggregate the results of queries against all possible histories (past or future, as applicable) within the limits specified by n. In grammatical terms, they involve the evaluation of the relevant quantities over all sentences that can be generated by the dynamic grammar. The algorithms for this reasoning are essentially algorithms implementing modal semantics, superimposed on top of standard formal language parsing and
CHAPTER 6: D'FOUNDATION - A COMPUTATIONAL FRAMEWORK

generation algorithms operating on sets of objects. They are briefly outlined later.

6.8 Dynamic Query and Update Processing Algorithms

ATNs have been used for natural language parsing and generation. Consequently, the available algorithms for ATNs (see for example [13], [25]) are characterized by two aspects:

- Specificity to natural language parsing. For example, registers such as "The noun of" are inherent adjuncts to nodes. Locus-search or bottom-up algorithms refer to and manipulate these specific registers, which are of no use to dynamic modeling.
- Most are based on exponential top-down or bottom up schemes since they sacrifice efficiency for conceptual and intuitive plausibility and expressiveness (the CYK-based algorithm we propose uses dynamic programming).

Due to these reasons, we choose to adapt widely applicable, easy to explain, yet reasonably efficient formal language parsing algorithms as a basis for this thesis. We do not claim that these are the optimal algorithms for the required functionality. Our main contribution remains the modeling of dynamics. Through these algorithms, however, we back our claim that the modeling devices which we have presented can be realized reasonably efficiently with the support of the data structures in chapter 7.

It is also beyond the scope of this thesis to propose and analyze all algorithms for each operation above. We introduce the links to the core algorithms underlying the operations and discuss variations and enhancements on this core briefly. The grammatical description of object dynamics is used in two ways. It is used as a history generator when a reaction is triggered by an attribute update and when effecting pre- and post-condition checks and actions associated with a reaction. Since a single grammar production is involved, the performance of generator algorithms is dominated by the class definition language and type of functions allowed, not by the type of grammar. Examples of the generative use of dynamic domain descriptions occurs in DML commands (3) and (5) as a rule, and in (4) and (7) if the command requires projection of existing object states into the future with respect to a given time. It is used as a recognizer by algorithms which parse the object history in order to infer its history-dependent attributes. This use is found in DML commands (1), (2) and (6), and in commands (4) and (7) if the command requires a projection of an existing object-state relatively into the past. (8) may use the grammar as a generator or recognizer.
6.8.1 Properties of Dynamic Grammars and their Recognizers

Grammars and recognizers, as well as the position of ATNs in the linguistic hierarchy, are discussed briefly in appendix C. Here, we review some particulars of the recognizers of dynamic grammars which are dictated by the nature of planning and dynamics.

In contrast to formal language theory, non-determinism in the parser for the ATN grammar describing dynamic behavior, and ambiguity in the grammar itself, are welcome properties of dynamic and adaptive information systems, even if the language of possible migration paths for an object is not inherently ambiguous [79]. If the grammar for a language (or the language itself) is ambiguous, then a stored history can legitimately have more than one parse tree. Ambiguity is necessary for describing real-life processes in order to admit many ways of reaching the same result in a flexible, recoverable process.

In the parser, non-deterministic is indicated by the necessity to consider more than one parse after consuming a and b." Concurrent branch" carries a different semantics, as given in this thesis.

FIGURE 6.16 Non-determinism and Ambiguity

On encountering a particular symbol, the state machine of a parser can non-deterministically be in two or more states, denoting two or more possible parses of the sentence, given the parser's knowledge at that point of the parse (its state). The need for non-deterministic choice in a parser arises if two sentences share a common prefix, and the grammar indicates the possibility of two distinct derivation trees for distinct strings with the given prefix. In terms of grammars describing dynamic behaviors, non-determinism in the parser reflects the possibility of an object having evolved along distinct paths specified by distinct derivation trees at any given time in its history (in retrospect). For context-free and context-sensitive histories ("sentences" of object-states), grammars with deterministic and non-deterministic parsers cover distinct classes of languages. Since both classes are of interest, the ATN interpreter in d’foundation should be capable of non-deterministic behavior.
6.8.2 High-Level Description of Operations on Dynamic Domains

The operations (1) through (8) discussed earlier call generative or recognitive algorithms acting on the grammatical definition of dynamic behavior, as discussed briefly below. We do not develop, optimize or analyze algorithms for the operations listed in section 6.7 as a whole, because optimization is not the main focus of this thesis. Instead, we introduce the concepts, and point to the use of the algorithms for history-based operations proposed in section 6.7.1, and the proposed data structures in chapter 7, as foundations underlying operations on dynamic domains. Operations such as (1) and (2) on recorded histories rely mainly on data structures:

**Type (1) Queries on temporal propositions**

SELECT-WHERE clauses can be evaluated using standard relational or object database query evaluation techniques on sets, using nested-inherited indices [22] for propositions involving aggregations or inheritance. This clause may be evaluated on a set of objects selected based on temporal or historical criteria specified by the:

- FROM <Set> ATDURING{(IMMEDIATELY) BEFORE|AFTER} <Interval> | <Temporal expression> clause, and/or
- FROM <Set> IMMEDIATELY (PRECEDING | FOLLOWING) <Class> clause and/or
- FROM <Set> G_SOURCE <Class> clause.

The sequencing of these selections based on query criteria to optimize the processing is outside the scope of this thesis. However, each of these clauses involve selection using our temporal data structures as follows:

- **<Temporal expression> (WHEN <Condition (Object)>)** evaluates to an interval which can be applied in other temporal selections. Temporal expression evaluation maps onto the operation: \[ \pi_x(\text{Start}(\text{Interval}), \text{End}(\text{Interval})) \sigma_{\text{Condition}} \text{Object} \]

This expression denotes the projection of the temporal interval associated with the object state resulting from applying the selection criteria <Condition> on Object. Both operations apply directly to the data structures, as in chapter 7.

- The ATDURING{(IMMEDIATELY) BEFORE|AFTER} clause translates directly to a data structure operation as well. The \( \sigma_{\text{Time}} \text{Set} \Rightarrow \text{Set}' \) operation applicable on our data structure addresses this functionality.

- (IMMEDIATELY) PRECEDING | FOLLOWING <Class> implies a selection based on the existence of a legal historical relationship of the object-states in <Set> to the state of being in <Class>. The set of object states immediately preceding or following a given state is found by using the \( \eta_{\text{time1}, \text{Class}} \text{Object} \Rightarrow C_1 \) and \( \gamma_{\text{time1}, \text{Class}} \)
Object \rightarrow C_2$ data structure operations discussed in chapter 7. Inverted indices on these attributes (transition-scope indices in section 7.5.2.3) may be used to evaluate this selection.

- The G_SOURCE $<$Class$>$ clause specifies a subset of $<$Set$>$ such that its members were generated from objects in $<$Class$. The generation-scope indices discussed briefly in section 7.5.2.3 may be used to perform this selection efficiently.

In the absence of generation-source indices, the evaluation of the G_SOURCE clause requires the parsing of each $Id(O)$ for $O \in$ $<$Set$. If $id(O)$ is defined by a constructor function $f.f^{-1}(Id(O))$ is used to determine the creator object's identity.

**Type (2) Queries on Historical propositions**

The examples of selection functionality in queries of type (1) above show the direct use of the data structure operations proposed in chapter 7. More complex queries exemplified by (2) also rely primarily on data structures, but may require parsing. (2) extends history-based selection functionality by the ON_HISTORIES clause:

- **FROM $<$Set$>$ ON_HISTORIES $<$[a, b, c]$> requires matching the recorded histories of objects in $<$Set$>$ against the specified selection criteria on their recorded histories. The selection function applied by this clause yields a set of objects $<$Set$>'$ such that each object in $<$Set$>'$ has a recorded history isomorphic to $<$a, b, c$, as specified in the query criteria.

Informally, this clause requires the following steps, executed iteratively, such that the remaining objects in $<$Set$_{i}>$ constitute the result from applying the selection clause.

1. $i \leftarrow$ length( <Historical_pattern>); $<$Set$_{i}> \leftarrow$ $<$Set$>$
2. $\forall o \in$ $<$Set$_{i}>$, apply $\gamma(x, \text{Historical_pattern}[i])$ $o \rightarrow C_2$ to find the previous class of $o$. $<$Set$_{i-1}>$ $\leftarrow$ $o$ if $C_2 = \text{Historical_Pattern}[i-1]$.
3. $i \leftarrow$ $i-1$; repeat (2)

The example above assumes that the members of set belong initially to the last class in the specified historical pattern. If they belong to the first, an analogous algorithm using $\eta_{(x, c_1)}$ Object $\Rightarrow C_2$ can be specified. More complex queries are easily extrapolated on this model.

**6.8.2.1 Introducing Grammatical Operations**

Though they rely primarily on data structures, the queries above may require the use of parsing algorithms. For instance, the SELECT $<<x>$ clause in (1) and (2) may be implemented by fetching a stored value in the simplest case. If a definition is found instead of a stored value, the evaluation of SELECT requires the evaluation of the function defining $<<x>$
through reasoning with dynamic relationships. For example, \(<x>\) may be defined by constructor functions on object-states in the object’s derivation history (figure 6.17). To calculate \(<x>\), \textit{PARSE}_{RECORDED}_{HISTORY} (section 6.8.4) is called to parse the recorded history.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure6.17.png}
\caption{Implicit Use of Dynamic Definitions in Parsing mode for Queries}
\end{figure}

The remaining operations use the ATN definition more directly:

**Type (3) Update Operations**

In the simplest case, the \textit{UPDATE} operation (3) may involve storing a new value interval through data structure operation \textit{Assert} \([\text{Attr Name} (\text{Obj Id})] --> \text{Value} | \text{Time Interval}]\). However, if the update triggers a change in dynamic status, it will implicitly use the ATN-based definition to generate future states. In general, therefore, \textit{UPDATE} may be described as:

$$\text{Update}(O.a=X_1, O.a=Y_{t'}) \Rightarrow \{C_1:O_{1t}, ..., C_n:O_{nt}\} \text{ and } \{D_1:P_{1t}, ..., D_m:P_{mt}\}$$

where \(O_i\) and \(P_i\) are objects and \(C_i\) and \(D_i\) are classes, and \(t\) immediately precedes \(t'\).

In this case, the function \textit{CHECK}_REAC (section 6.8.3) is called, essentially to execute the reasoning outlined in section 6.3 and 6.4, depending on the type of reaction triggered.

**Type (4) Class-centered Update Operations**

The \textit{CHANGE}\_\textit{CLASS} operation (4) specifies the establishment of a dynamic relationship between two object states. This may imply multiple reactions and the establishment of dynamic relationships among related object-states. For example, given

\[ \text{A} \rightarrow B.C \text{ and } \text{B} \rightarrow D.E \]

changing the class of an object in \(A\) to \(E\) would need more than one reaction. If more than one sequence of reactions can lead to \(E\), selection must be performed. If object-states \(A\) and \(E\) are aggregated from (into resp.) other object states, each sub- (super- resp.) aggregate object may undergo a reaction as a result of \(A\) evolving to \(E\). \textit{CHANGE}\_\textit{CLASS} therefore involves the use of the ATN description in a generative role. \textit{CHANGE}\_\textit{CLASS} (Source, Target) requires the following high-level operations:
• Call \textit{Find-Path}(Source, Target), a \textit{data dictionary} operation which can be pre-compiled (section 6.8.5) to return a set of sequences of grammar productions \(<p_1 \ldots p_n>\) such that \emph{Source} \in \text{lhs}(p_1) and \emph{Target} \in \text{rhs}(p_n) and the \text{lhs} of every \emph{p_i} has a symbol that belongs to the \text{rhs} of \emph{p}_{i-1}. In other words, returns a sequence of reactions linking \emph{Source} to \emph{Target}.

• (Till success) Select a path. (Selection logic may vary; not discussed in this thesis).

• For every reaction in the selected path, call \textit{EFFECT_REAC} (section 6.8.3).

\textbf{Type (5) Reaction-related Triggering Operations}

The \textit{TRIGGER} operation (5) specifies a similar operation as (4), but in terms of a reaction which must be executed at a given point in time. The high-level description of this operation is similar to the \textit{CHANGE_CLASS} operation.

\textbf{Type (6) Reversal of Material Actions}

The \textit{REVERSE} operation (6) asks for parse of the derivation history of an existing recorded history to be reversed, to restore the recorded history to a previous sentential form. If the recorded history is \(a_1,a_2,\ldots,a_n\), the reverse operation invoked on state \(a_n\) may simply mean a reduction in length to \(a_1,a_2,\ldots,a_m\), where \(m<n\) in the simplest case, but may lead to the replacement of the history by a sequence of object-states \(a_1,a_2,\ldots,b_1,b_2\) in another case where the reversal of a grammar production involving non-terminals is involved. The implementation of this operation requires the partial execution of the \textit{PARSE_RECORDED_HISTORY} function (in a re-entrant fashion so that intermediate states can be inspected) until the required state is reached. The high level process is therefore:

• Execute one step of the \textit{PARSE_RECORDED_HISTORY} algorithm.

• If desired state not reached, continue, else stop.

\textbf{Type (7) Goal-Directed Updates}

The \textit{TRANSFER} operation (7) combines the use of the dynamic domain definition as a generator and parser through goal-directed reasoning. It implements the following function:

\textit{Transfer}(C_1:O_1 \mid t', C_2:O_1 \mid t) = \Rightarrow C_2:O_1 \mid t', D_1:P_1 \mid t_1, \ldots, D_n:P_n \mid t_n

where \(C_i\)'s and \(D_i\)'s denote classes, \(O_i\)'s and \(P_i\)'s denote objects, and \(t_1\)'s may temporally precede or follow \(t\) and \(t'\), which may relatively precede or follow each other in time.

• the left-hand-side expression states the requirement to achieve \(C_2:O_1 \mid t'\) given \(C_1:O_1 \mid t\), i.e make \(O_1\) a member of class \(C_2\) at time \(t'\), given the fact that it is a member of class \(C_1\) at time \(t\). It may relatively precede or follow \(t'\).
Transfer may achieve this objective in the manner indicated by the right-hand-side expression, i.e. achieve \( C_2:O_1 | t' \), in the context of \( D_i : P_i | t_i \), the state of \( n \) objects \( P_i \) being in \( n \) corresponding states \( D_i \) at various times \( t_i \).

At a high-level, transfer may be executed as a goal-driven state-space search problem (see [37]), where a state \( S_1 = Class1: Object | time1 \) is said to succeed state \( S_2 = Class2: Object | time2 \) if there is a reaction to take \( Object \) from \( Class1 \) to \( Class2 \). A state \( S_1 = Class1: Object | time1 \) is said to be reversible_to \( S_2 = Class2: Object | time2 \) if there is a reaction which is reversible and which takes \( Object \) from \( Class2 \) to \( Class1 \). We use \( succeed^*(X) \) and \( reversible_to^*(X) \) to denote the transitive closure of the succeed and reversible_to relations with respect to state \( X \). Note that \( succeed^* \) and \( reversible_to^* \) relationships can be precompiled based only on the definition of the ATN grammar without considering the pre- and post-conditions.

If the target state is in the future with respect to the source state, we first check if it can be reached through repeated application of the succeed function. Failing that, we see if we can reverse to a state from which \( C_2 \) can be reached by repeatedly applying succeed, and failing that, we check if we can reach a state in the future with respect to \( C_2 \), from where we can reverse to \( C_2 \). The outline below expresses this sequence.

1. Initial state = \( C_1: O_1 | t \), Final state = \( C_2: O_1 | t' \).
2. If \((t < t')\) then If \((C_2 \in succeed^*(C_1))\) then
   
   \[ CHANGE\_CLASS\ C_1\ to\ C_2;\ \text{if unsuccessful, go (3)} \]
3. Find classes \( R \) such that \( R \in reversible_to^*(C_1) \), and \( C_2 \in succeed^*(R) \), do
   
   \[ REVERSE \ O_1 \ TO \ R; \ CHANGE\_CLASS \ O_1 \ to \ C_2; \ \text{if unsuccessful go (4)} \]
4. Find classes \( S \) such that \( S \in succeed^*(C_1) \), and \( C_2 \in reversible_to^*(S) \), do
   
   \[ CHANGE\_CLASS \ O_1 \ to \ S; \ REVERSE \ O_1 \ TO \ C_2; \ \text{if unsuccessful go (4)} \]

If \( C_2 \) is in the relative past with respect to \( C_1 \), we first try to reverse directly to it, failing which we may try to generate some history in the future from which we may reverse to \( C_2 \), or reverse beyond \( C_2 \) in the past and try to generate forwards to \( C_2 \). We abbreviate below:

5. If \((t' < t)\) then If \((C_2 \in reversible_to^*(C_1))\) then \( REVERSE \ O_1 \ to \ C_2. \) If unsuccessful, try (4) followed by (3) above in this sequence.
6. If neither \((C_2 \in reversible_to^*(C_1))\) nor \((C_2 \in succeed^*(C_1))\), then try steps (3) followed by (4) in this sequence.

The outline above uses a depth 1 search heuristic. If the target class is not directly
reachable while generating forwards, it tries to backtrack (or generate forwards while backtracking) only once. A complete algorithm would attempt backtrack (generation) on each failure. We do not survey search algorithms in this thesis, but through the outline above, clarify the relationship of TRANSFER to the core grammar based algorithms.

**Type (8) Modal Queries**

Necessity under the depth-restricted modal framework we have proposed is interpreted as truth in all relevant states reachable within \( n \) steps, where \( n \) is the limiting parameter introduced earlier. Similarly, possibility is interpreted as truth in at least one relevant state reachable in \( n \) steps. The overall algorithms are obvious, and outlined as follows:

1. If the time associated with \( \text{Proposition} \) is \( < t_{\text{now}} \), \( F \leftarrow \text{PARSE}_{\text{RECORDED\_HISTORY}} \)
2. If time associated with \( \text{Proposition} \) is \( > t_{\text{now}} \), \( F \leftarrow \text{EFFECT\_REAC}(X) \), where \( \{X\} \) denotes the set of all reactions sourced at the current node.
3. \( \text{Currently\_Evaluated\_Propositions} \leftarrow \emptyset \)
4. For counter = 1 to \( n \) do
   
   \( \text{call } F \text{ on current recorded history and state; } S = \text{newly created states} \)
   
   \( \text{Currently\_Evaluated\_Propositions} \leftarrow \text{ESTABLISH \text{Proposition} on relevant states in } S \);
5. If operation = \text{NECESSARY}, return( \( \land P, \text{such that } P \in \text{Currently\_Evaluated\_Propositions} \))
   
   else \( \text{return( } \land P, \text{such that } P \in \text{Currently\_Evaluated\_Propositions} \) )

This completes our discussion on the relationship of the operations to the core grammar-based and data-structure-based functions that are necessary for dynamic and adaptive information systems, having earlier showed how the d'dialect maps onto ATNs, and how ATNs relate to CFG's with annotations. Hereafter, we discuss the algorithms using the terminology, viz. states and transitions, of conventional formal language for clarity.

**Notation:** In the algorithms, we use **Bold** to indicate data structure operations (operation names from chapter 7 are mentioned as comments, *Italics* for data dictionary operations, and *superscripts* to mark points for later analysis. Raise_Exception() captures all aspects of notification and rollback in case of error.

### 6.8.3 Effecting Reactions: The ATN Definition as a Generator

Operations which effect reactions directly or indirectly, in a committed or a hypothetical manner, use the ATN as a generator. In this role, the ATN contributes a precise definition
of the allowable changes in a domain, usable for top-down generation of histories. In the pre-
ceeding sections we have referred to algorithms CHECK_REAC and EFFECT_REAC. These algo-
rithms implement the candidature checking, attribute updating, and token propagation logic
given in sections 6.3 and 6.4. There are algorithms for forward generation of natural lan-
guage sentences using ATNs, but they do not model their domain in terms of Definitional/
Factual Attributes, calculate Dormant/Expiring Attributes, or trigger reactions based on incremen-
tal updates. These needs are particular to dynamic and adaptive information systems; thus,
we must adapt ATN concepts. Analysis of these algorithms is done in 6.8.3.1 and 6.8.3.2.

The algorithm CHECK_REAC is used whenever an object becomes a candidate (section
6.3) to decide and control which (if any) reaction(s) is undergone by an object obj after
attribute attr is updated to value val, and to effect one such reaction (extendable to n).

\[
\text{CHECK_REAC}(\text{obj, attr, val, time})
\]

1. Get attr(obj) | \( I, \) Covers(I, time) /* \( \pi(\text{Attr, Time}) */
2. Set attr(obj) \( \gets \) val | [time, End(I)] /* Assert [ Attr(Obj) \( \gets \) Val][\text{Interval}] */
3. If (Error) \{ Raise_Exception() \}
4. else \{ Find \( c_i \) s.t. \( \text{obj} \in c_i, \quad C := \{c_i\} \); /* \( \pi \text{Obj, Time} \Rightarrow \text{Collections(O)} */
5. \( V := \Phi \);
6. \( \forall c_i \in C \) do \{
7. \quad \text{Find_Links} \quad r_i \) s.t. \( r_i \in \text{Outlinks}(c_i) \); \( R := \{r_i\} \)
8. \quad \forall r_i \in R \) do \{
9. \quad \quad \text{If attr } \in \text{Pre_Conds}(r_i) \{
10. \quad \quad \quad \text{If } (\text{Pre_Conds}(r_i) = \text{True})^{(1)}
11. \quad \quad \quad \quad \{ V := V \cup <\text{obj}, c_i, r_i, \text{target}(r_i), \text{time}> \} 
\}
12. \quad \quad \}
13. \quad \}
14. \quad \text{If } (|V| = 1) /* Unambiguous reaction is indicated */
15. \quad \{ \text{EFFECT_REAC}(V) \}
16. \quad \text{else } \{ \text{DECIDE_REAC}(V) \} /* >1 possible reaction */
\}
17. \text{Candidate(obj) } \gets \text{False}

The arguments of the following EFFECT_REAC algorithm indicate the intention of
making the needed updates to make object obj traverse link reac from node src to node tgt at
time \( time \). \( R \) is the set of registers associated with \( obj \) prior to the reaction. \( \text{EFFECT-REAC} \) is called after selecting a reaction among those whose pre-conditions are satisfied. The algorithm starts by finding the set of ATN links interpreting the reaction. In the case of a complex reaction such as a production, this may be a "list of sets of ATN links". In terms of our example in section 6.3.4, the reaction \( \text{SO} \rightarrow \text{MO} \) maps onto \([\{<\text{NSO}, \text{NSO}>\}, <\text{NSO}, i>\], <\text{IO}, \text{MO}>\], \) where [] denotes lists and {} denotes sets.

\[
\text{EFFECT-REAC}(<\text{obj}, \text{src}, \text{reac}, \text{tgt}, \text{time}>))
\]

1. Accept user input \( U \) as a set of \(<\text{reg} \leftarrow \text{val}>\) declarations.
2. \( L = \text{Find_ATN_for_Reac(r)} \) /*setof ATN-links interpreting reaction \( r \) */
3. If \(|L| > 1\) \{ \text{START TRANSACTION} \}
4. If \( \text{Generation}(\text{reac}) \) \{
   \begin{enumerate}
   \item \( \forall a \in \text{Defn_Attr}(\text{tgt}), \text{Define (nttgt:regunta)} \)
   \end{enumerate}
5. Let \( L = [h|T] \)
6. If, \( h = \phi \), then \{ \text{END TRANSACTION; return} \}
7. Else \{ \text{If Token_Identification_Actions(\( r \))}
8. \( \text{Establish(Token_\text{Id} Actions, \phi, \phi)} \}
9. \( \text{Registers = \cup \{ (\cup_j \text{ ntSource}(l_j):a_j) \cup (\cup_k \text{ ntTarget}(l_k):b_k) \} \)}
   \begin{align*}
   \text{where } l_j & \text{'s are the ATN links in } h, a_j \text{'s represent all} \\
   \text{attributes of Source(l_j), and } b_k & \text{represent all definitional} \\
   \text{attributes of Target(l_j).} 
   \end{align*}
10. \( \forall \text{reg} \in \text{Registers do} \{
   \begin{enumerate}
   \item \text{Establish(reg(\text{Token})|_time, U \cup \text{Registers, Conditions(reg)})}^{(2)}
   \end{enumerate}
   \}
11. \( \forall \text{a in Actions(l) do} \{ \text{Establish(a, U, Conditions(a))} \}
12. \( \text{If Concurrent_Branching(r) \{Establish(Concurrent_Branch_Actions, \phi, \phi)} \}
13. \( \forall a \in \text{Actions(l) do} \{ \text{Establish(a, U, Conditions(a))} \}
14. \( \text{If } (-\text{Post_Conds(l)}|_time^{(4)}) \text{ Raise_Exception()}, \text{go to } 1. \)
15. \( L \leftarrow [T], \text{go to } 6. \)
\)

\( \text{EFFECT-REAC} \) calls upon \( \text{ESTABLISH} \) to perform or verify certain calculated attributes. \( \text{ESTABLISH} \) is an umbrella algorithm for managing the update of attribute values under the constraints imposed by the relevant postconditions. If no update is needed, the value is verified to obey the conditions on an attribute. Establishing a given predicate on an attribute consists of storing the value (after calculating it if it is an evaluated attribute), or storing the function itself if it is defined by a constructor function.
**ESTABLISH** \( (n_{\text{class}}:\text{reg}(\text{Token}) |_{\text{time}}, \text{Available_Vals}, \text{Conditions}) \)

If \( n_{\text{class}}:\text{reg}(\text{Token}) \in \text{Expiring Attr}(\text{Class}) \):

If \( n_{\text{class}}:\text{reg}(\text{Token}) \in \text{Eval Attr}(\text{Class}) \):

1. If \( (n_{\text{class}}:\text{reg}(\text{Token}) = f(\phi), \text{i.e. is a simple attribute}) \) \{ 
   2. If \( (<n_{\text{class}}:\text{reg}(\text{Token}), \text{value}> \in \text{Available_Vals}) \)
   3. \( \text{Set } a \leftarrow \text{Value | time} \)
   4. Elseif \( (n_{\text{Source}(\text{li})}:\text{reg}(\text{Token}) \in \text{Available_Vals}) \)
   5. \( \text{PROPAGATE } (n_{\text{class}}:\text{reg}(\text{Token}) \leftarrow n_{\text{Source}(\text{li})}:\text{reg}(\text{Token}) ) \)
   6. Else \{ Prompt the user for the value \}

7. Else \{ /*n_{\text{class}}:\text{reg}(\text{Token})=f(a_i |_{\phi}), i = 1..n \text{ is an evaluator */} 
   8. \( \forall a_i |_{t_i} \text{ which are parameters of } f(a_i |_{\phi}) \text{ do } \) 
   9. \( \text{If } (p_i \leftarrow \text{Get } a_i |_{t_i} \text{ is not successful } ) \) \( p_i \leftarrow \text{DERIVE}(a_i |_{t_i}) \)

10. \( p \leftarrow \text{evaluate}(f(a_i |_{\phi}), i = 1..n) \)
11. \( \text{Set } a \leftarrow p |_{\text{time}} \)

on failure at any step \( \text{Raise_Exception()} \)

If \( n_{\text{class}}:\text{reg}(\text{Token}) \in \text{Constr Attr}(\text{Class}) \):

12. \( \text{Set } n_{\text{class}}:\text{reg}(\text{Token}) \leftarrow "f( a_i |_{\phi} )" |_{\text{time}} \) 

on failure, \( \text{Raise_Exception()} \)

If \( n_{\text{class}}:\text{reg}(\text{Token}) \in \text{Dormant Attr}(\text{Class}) \):

13. If \( (\text{Prev Membership}(\text{Token}, \text{Class})(5) = Q) \) \{ 
14. \( \text{Set } n_{\text{class}}:\text{reg}(\text{Token}) |_{\text{time}} \leftarrow n_{\text{class}}:\text{reg}(\text{Token}) |_{\text{End}(Q)} \)
}
15. Elseif \( (\text{DERIVE}(n_{\text{class}}:\text{reg}(\text{Token}) |_{\text{End}(Q)} \neq \phi) \text{ where } Q \text{ precedes } \text{time} \)
16. \( \text{Set } n_{\text{class}}:\text{reg}(\text{Token}) |_{\text{time}} \leftarrow n_{\text{class}}:\text{reg}(\text{Token}) |_{\text{End}(Q)} \)

otherwise \( \text{Raise_Exception()} \)

Get \( n_{\text{class}}:\text{reg}(\text{Token}) |_{\text{time}} \)

\( \text{VERIFY}(n_{\text{class}}:\text{reg}(\text{Token}) |_{\text{time}} \text{ against Conditions}) \)

The functions \text{Get} and \text{Set} are purely data structure operations in the sense of a fetch or storing data. They map onto the operations in chapter 7 as shown under the \text{CHECK REAC} algorithm. The function \text{DERIVE} is used in parsing a history using the grammatical definition of dynamic behavior as shown in the algorithm \text{PARSE RECORDED HISTORY} in section 6.8.4 to derive a specific state, and extract the value of an attribute associated with the derived state.
If the pre-conditions of more than one reaction are satisfied, making them candidates for traversal, `DECIDE_REAC` ranks them in the order of preference. Preference is judged according to the heuristic given in section 6.3.3.1- the reaction whose target class is closest in terms of definition to the current object-state is naturally preferred. This algorithms only aids a user and is not mandatorily. The preferred reaction is triggered using `EFFECT_REAC` as illustrated above.

**DECIDE_REAC(V)**

1. Selection_Tree <- φ
2. ∀ candidate = <obj, src, reac, tgt, time> ∈ V do {
   3. calculate d = DIFF(Obj_State(obj), Defn(tgt)) \(^{(6)}\)
   4. Sort (Selection_Tree ∪ <d, candidate>)
}
5. q = Choose(Selection_Tree) /* Ask user /select top candidate */
6. EFFECT_REAC(q)

**CHANGE_CLASS** is used to migrate an object in class `src` to an object-state in class `tgt` by sequentially effecting a series of reactions which are pre-stored. Though more general run-time algorithms for generating strings top-down exist in the established literature (see [79] or another text on compiler construction), we assume pre-compilation for efficiency.

**reentrant CHANGE_CLASS** (o, s, p, t)

1. P = Find_Path(s, t)
2. If (P = φ) {
   3. Raise_Exception()
}
3. else { Start <- s
4. While P ≠ [] do {
   5. let P = [h | T] /* point_of_reentrance */
6. EFFECT_REAC(o, Start, h, Target_Class(h), time)
7. Start <- Target_Class(h)
}

**CHANGE_CLASS** relies on the data dictionary function `Find_Path`, which stores precompiled paths. We omit details regarding how the time parameter may be manipulated in a multi-step reaction sequence, since this is essentially user-dependent.

The **DERIVE** function is used to parse the recorded history backwards, or to generate forwards from the last recorded object state, until the desired state is reached. The logic of
this function consists of extending the `PARSE_RECORDED_HISTORY` algorithm by examining the states assigned to the `STATE` table at the points marked by the word `point_of_reentrance`. If the required state is reached, the algorithm is terminated. Generating forwards is a similar extension of the `CHANGE_CLASS` function. We omit details of `DERIVE` for brevity.

Example 6.6 Use of some generator algorithms for actively triggered updates in a production control support system example (section 6.3.4).

The classes below were introduced in the paper production example:

- **Sales_Order** (SO): `(Reco_Date ≠ ∅ ∧ Deliv_Date ≤ 1/1/2000 )`

- **Matl_Order** (MO): `(Matl_Date ≤ 1/1/2000)` where `Matl_Date = Source.Deliv_Date - 10`

The reaction linking these classes is `SO-/>-> MO`. We assume that there exists an object O which is a sales order at time t, represented by token `To` resident at node `nSO`, s.t. `nSO: rReco_Date(To) = 10/1/1991`, and `nSO: rDeliv_Date(To) = 10/1/1995`.

- **UPDATE** O. `Candidate TO True AT 9/25/1995` is invoked to explicitly trigger a reaction. This operation leads to the call `CHECK_REAC(O, Candidate, True, 9/25/1995)`.

- **CHECK_REAC**(O, Candidate, True, 9/25/1995) sets the value of Candidate to True, and calls `Find` to determine the present membership of O, which is SO.

- calls `Find_Links` to discover the links leaving `nSO`. There are 3 such links: `<nSO, nCol>`, `<nSO, nSSO>`, and `<nSO, nMO>`. All of their preconditions are true since `nSO: Candidate = True` is a common precondition. Thus, `V <- {<O, SO, SO-/>->SSO, SSO, 9/25/1995>, <O, SO, SO-/>->Col, Col, 9/25/1995>, <O, SO, SO-/>->MO, MO, 9/25/1995> }`.

- Since `|V| > 1`, calls `DECIDE_REAC(V)`

- **DECIDE_REAC**(V) sorts the three candidate reactions with respect to the distance from the current object state. In this case, the target classes SSO and MO are equidistant from the current object-state (there is one discrepancy each between the definitions of SSO and MO, and the current attribute values of O). Assume that the user chooses to execute `SO-/>->MO`. `EFFECT_REAC(SO-/>->MO)` is called.

- **EFFECT_REAC**(SO-/>->MO) uses `Find_ATN_for_Reac(SO-/>->MO)` to retrieve the set of ATN links interpreting this reaction, which is `[[<nSO, nSO>, <nSO>], <1, nMO>]]`.

  - a transaction is started
  - since SO-/>->MO is a generation, the new register `nMO: rMatl_Date` is defined.
  - for the set of links `{<nSO, nSO>, <nSO>}`, `Registers <- {nSO: rReco_Date(To), nSO: rDeliv_Date(To), n: rReco_Date(To), n: rDeliv_Date(To), n: rMatl_Date(To)}`. The three significant calls are:
    - `ESTABLISH(nSO: rReco_Date(To)|9/25/1995, Registers, nSO: rReco_Date(To)|9/25/1995 ≠ ∅)`
    - `ESTABLISH(nSO: rDeliv_Date(To)|9/25/1995, Registers, nSO: rDeliv_Date(To)|9/25/1995 ≤ 1/1/2000)`
ESTABLISH \( n_i : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \), Registers, \( n_i : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \leq 1/1/2000 \) are made to find the values of the registers. The others are trivial.

- The Concurrent Branching actions as shown in section 6.3.5 are executed.
- for the remaining link \( <n_{MO}> \), the Token_Id_Actions associated with the link are executed as in section 6.3.5 to create a token \( T_{new} \) and assign the values of all registers to \( T_{new} \). Registers \( \leftarrow \{ n_i : r_{Mat\_Date}(T_o), n_{MO} : r_{Mat\_Date}(T_{new}) \} \).

ESTABLISH \( n_i : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \), Registers, \( n_i : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \leq 1/1/2000 \)

ESTABLISH \( n_{MO} : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \), Registers, \( n_{MO} : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \leq 1/1/2000 \) are called to find the values of these registers.

- the transaction is ended.

- **ESTABLISH** \( (regl\_time, \text{Available\_Values}, \text{Conditions}(regl\_time)) \) treats each case differently:

Some registers are initialized from their previous values:

- \( n_{SO} : r_{Recv\_Date}(T_o) \) \( 9/25/1995 \) is an expiring evaluated attribute. Without user input, it is initialized from its previous value from Available_Vals: \( n_{SO} : r_{Recv\_Date}(T_o) \) \( 10/1/1991, 9/25/1995 \)

- \( n_{SO} : r_{Deliv\_Date}(T_o) \) \( 9/25/1995 \) is similarly copied from \( n_{SO} : r_{Deliv\_Date}(T_o) \) \( 10/1/1991, 9/25/1995 \)

Some registers are calculated from non-trivial functions:

- \( n_i : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \) is an expiring evaluated attribute defined by the function \( \text{minus}(n_{SO} : r_{Deliv\_Date}(T_o) \) \( 10/1/1991, 9/25/1995 \), 10) \). The value of the register \( n_{SO} : r_{Deliv\_Date}(T_o) \) \( 10/1/1991, 9/25/1995 \) is found using data structure operations.

Some registers undergo a null operation:

- \( n_i : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \) and \( n_{MO} : r_{Mat\_Date}(T_o) \) \( 9/25/1995 \) are assigned by the token identification actions and retain their values at 9/25/1995.

The example above shows the use of the algorithms in enforcing the conditions related to the traversal of the links that interpret reactions. On its basis, we argue briefly for the completeness of the actions below.

**6.8.3.1 Correctness of Generation Algorithms**

In sections 6.3 and 6.4, we discuss the pre- and post- conditions and actions needed to interpret faithfully the semantics of the basic dynamic devices and adjectives, and argue for their correctness. The algorithms for using the ATN as a generator reflect directly the verification of these conditions and performance of these actions. Their correctness follows from the discussion in earlier sections. We show the correspondence of the individual steps in our algorithms to the semantic invariants of reactions and their interpretation in ATNs below.

**CHECK_REAC** identifies which reaction(s) may be triggered by an update, ensuring:

- An object may be a member of more than one class, and may undergo a reaction with respect to any of its current class memberships. Step 4 find all current mem-
emberships and considers reactions with respect to each current membership.

- **Reaction can be triggered explicitly by a user making an object a candidate, or implicitly when an update violates the source class definition.** Step 10 ensures this condition by checking the pre-conditions. All pre-conditions proposed in this thesis (sections 6.3.3.1, definition 6.3) are logically subsumed by the clause: 
  \[(n\text{Class} \cdot \text{Candidate}(T_o) = \text{True}) \lor (n\text{Class} \cdot \text{a}(T_o) \leftarrow \text{Val})\], which ensures that in every situation, either Candidate must be set explicitly, or an update has to occur.

- **All reactions are considered when the situation above occurs.** Steps 11 and 16 ensure that every reaction whose pre-condition is satisfied is considered.

- **After an object is considered for a reaction, it is not considered again until an external event occurs to make it a candidate again.** Step 17 ensures that the Candidate register is reset, whether or not a reaction is effected. The pre-conditions above then ensure that no reaction is considered until the next event occurs.

**DECI R REAC** is the procedure for selecting which of many candidate reactions should be triggered. Our framework proposes no rule for this decision. The correctness of this procedure is therefore not at issue. However, a heuristic for ordering the reactions is proposed:

- **Prefer the reaction whose target class p is closest to the current object state in terms of the number of definitional constraints of p satisfied.** Step 3 calculates the distance \(\text{DIFF}\) above. Step 4 sorts them in increasing order to meet this heuristic.

- **In case of contention, user chooses.** Step 5 implements this requirement.

**EFFECT_REAC** is the algorithm for performing the actions to ensure the integrity of a selected reaction. It’s responsibility is to ensure that the post-conditions specified are verified before a reaction is completed and all actions implied by type of reaction or adjectives are performed. Specifically, this task comprises:

- **Ensure that the interpretation of reactions through ATN links is preserved.** Steps 1 and 7 ensure transactional behavior. We assume that \(\text{END\_TRANSACTION}\) does nothing if no transaction is active.

- **If multiple ATN links are used to interpret a reaction, ensure that they are performed as a unit.** Sequences of concurrent ATN link traversals are allowed to interpret a reaction in step 2. Step 6 therefore processes lists of sets of ATN links.

- **If the reaction is a generation, establish distinct identities and an alternative set of registers.** Identification actions are supported in step 8, as specified in definition 6.4. These actions create a new token and transfer all registers to the new token. New registers are created in step 4 for all generations.

- **Ensure the concurrent branching conditions if an object must concurrently be in**
more than one class after a reaction. Concurrent branch actions of step 13 ensure continuity of membership if link source and target nodes are identical.

- **Ensure that other actions implied by adjectives are performed.** Step 14 performs actions such as register creation and update under constraints (example in section 6.4.2.4) which are not implied by source or target class definitions.

- **Verify the conditions imposed on registers by the link traversal.** All conditions are expressed as constraints on register values. Section 6.3.3.7 enumerates all register types, and constraints imposed on them by the type of reaction and the source and target classes. Step 12 ensures that the conditions are passed to \textit{ESTABLISH} for verification under update (see example 6.6).

\textit{ESTABLISH} is the function for updating and verifying individual attributes under the semantics of different attribute types imposed by our framework, as follows.

- **Determine values of Expiring Evaluated attributes.** Expiring evaluated attributes may be defined by a trivial function $f(\phi)$ or by a real function of attributes in other object-states. In the first case, the user may have specified a value (step 2), or the attribute may already have a value in the previous object state that can be propagated to the target state (step 4). If neither case holds, the user is queried (step 5). In the second case, the values of each of the function parameters (i.e. attribute values from other object-states) must first be found by database search (step 8), or by parsing the recorded history if the object-state has been replaced (step 8, if condition), followed by function evaluation (step 9). No other case exists.

- **Determine values of Expiring Constructed attributes.** Constructed attributes are not evaluated unless requested by a \textit{SELECT} operation. This case is therefore realized by a store definition function (step 11).

- **Determine values of Dormant attributes.** Dormant attributes are regenerated from their value during a previous membership in the same class. The record of previous membership may be stored (in which case, the previous membership is searched using the data structure operation $\Pi_{(\text{time1}, \text{Class})\text{Object}} \Rightarrow C_1 = \text{Collection(\text{Object})}_{\text{time}}$ applied successively (step 12). If the record of previous membership is not stored, it must be derived (step 14) by parsing the recorded history.

- **Verify the constraints on all attributes.** In any case, or even when an update is not done, the function must verify the validity of the value against the constraint to ensure system integrity. This is ensured by step 15.

The arguments above in conjunction with the discussion (sections 6.3 and 6.4) regarding translation of the intended semantics of the basic reactions and adjectives in terms of ATN constructs, argue for the completeness of our framework for generating histories based on ATN descriptions.
6.8.3.2 Complexity of Generation Algorithms

The complexity of reasoning with ATNs as a generator are presented below as propositions rather than theorems because our claims depend loosely on some simplifying assumptions made in the following informal proofs. We claim that although inherent worst case undecidability or exponential complexity is faced, many real cases with acceptable cost exist.

Proposition 6.4 The computational complexity of the algorithms above for generating histories triggered by updates to individual attributes is as follows:

1. If parsing of the recorded history is required, the computational cost is asymptotically bounded by the cost of parsing (proposition 6.5) * O(log(n)) in the size of the history.

2. In cases where the number of attributes is bounded and other simplifying assumptions hold, the complexity is bounded by O(log(n)) in the size of the history.

3. In case parsing recorded histories is not required, but there is a very rich domain in terms of classes, the computational cost is asymptotically bounded by O(n.log(n)) in the size of the history.

5. In the worst case, history generation is not computable. In the worst computable case, it can be intractable.

Informal Proof:

We summarize the reasoning behind proposition 6.4 in figure 6.18, and explain the reasoning behind the numbers as follows:

| Question                          | Symbolically | Worst   | "Realistic" | "Many Cases"
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Find and update all AVInts</td>
<td>O(log(v))</td>
<td>unbounded</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>Find all classes O belongs to</td>
<td>O(C) = O(v)</td>
<td>unbounded</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>Find reactions from each C</td>
<td>O(R)</td>
<td>unbounded</td>
<td>O(C^2)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Find qualifying reactions</td>
<td>O(C^R*log(C^R))</td>
<td>uncomputable</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>- sort on distance</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- no. of reactions chosen</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of attributes to update</td>
<td>O(a)</td>
<td>unbounded</td>
<td>O(1) or O(log(n))</td>
<td>O(1)</td>
</tr>
<tr>
<td>Cost of Updating</td>
<td>O(u)</td>
<td>uncomputable</td>
<td>O(n^3*log(n))</td>
<td>O(log(n))</td>
</tr>
</tbody>
</table>

Number of attributes (a) is theoretically unbounded  O(log(n)) -> O(1)
Number of AVInts (v) is theoretically unbounded      O(n)
Number of classes (C) is theoretically unbounded     O(1)
Number of reaction (R) is theoretically unbounded    O(1)
Cost of evaluating an attribute is theoretically unbounded, unrelated to history size, usually simple O(1)

FIGURE 6.18 Complexity Estimates for Generator and Parser algorithms

reasoning behind the numbers as follows:

CHECK_REAC manages the decision regarding whether an update triggers a reaction. It first fetches and updates all the updated attributes. The complexity of this operation is:
Theoretically unbounded, because the number of AVInts (v) to be updated is not strictly related to the history size.

Usually, we are justified in making the assumption that \( v = O(n) \), where \( n \) is the size of the history, since attributes are not updated in an unlimited way.

As chapter 7 shows, the cost of is \( O(\log(n)) \) in the size of the history to search and update \( O(n) \) AVInts. \textit{CHECK\_REAC} then finds the current class memberships of the object.

- Theoretically, the number of classes is unlimited. An unlimited number of classes can be defined on the basis of even a single attribute by subdividing its domain.
- In practice, we can assume \( n_c < n_a < n \) where \( n_c \) is the number of classes, \( n_a \) is the number of attributes and \( n \) is the size of the history. Since \( O(\log(n)) \) is the cost of finding the collections of an object at an arbitrary time and \( O(1) \) is the number of classes, \( O(\log(n)) \) is the figure used.

For each class, the number of reactions incident are found.

- In theory, the number of reactions between \( n \) classes is unbounded, since there can be any number of named reaction between any two classes.
- We may expect \( O(C^2) \) reactions among \( C \) classes if the domain is well connected. More often, the number of reactions \( R = O(C) = O(1) \). \textit{Find\_Links} also costs \( O(1) \).

Thereafter, the cost of \textit{CHECK\_REAC} is dependent on the number of classes which the object belongs to, and the number of reactions incident on each class. \( O(n_a) \* O(C*R) \) preconditions may need to be evaluated. \( n_a = O(1) \) as discussed earlier.

- Note (1): Pre-conds can be theoretically uncomputable (e.g. Turing Machine \( T \) halts on input \( a \)).
- However, we assume the conditions are simple (e.g. \(<, >, = \) checks, which are evaluable in \( O(1) \) time each.

Thereafter, the cost of \textit{CHECK\_REAC} is dominated by the cost of \textit{DECIDE\_REAC}.

\textit{DECIDE\_REAC} must first calculate the \textit{DIFF} function for, then sort and order up to \( O(C*R) \) reactions.

- Calculation of \textit{DIFF} involves \( O(n_a) = O(1) \) comparisons of attribute values between current object state and target class definition. If each comparison is \( O(1) \), the cost is also \( O(1) \) under our earlier assumptions.

Note (6): The distance between any object state and a class description can be calculated in unit cost, based on the summary descriptors (section 7.4.1). Thereafter, the cost of \textit{DECIDE\_REAC} is dominated by \( O(1) \) calls to \textit{EFFECT\_REAC}.

\textit{EFFECT\_REAC} finds the set of ATN links (which is \( O(1) \)) corresponding to the reaction and performs all actions necessary for each link. First, \( O(n_a) = O(1) \) new registers may be
defined. Following that, \( O(n_a) \) registers are established, i.e. \( O(n_a) \) assignments are carried out under prescribed post-conditions. This includes Token_Identification and Concurrent_Branch actions, which are both bounded.

**Note (4)**: As in the case of pre-conditions, post-conditions may be theoretically uncomputable, but in general, we can assume them to be simple functions of \( O(1) \) complexity.

`ESTABLISH` behaves differently depending on the type of attribute.

- Theoretically, actions can be uncomputable; if computable, the complexity of each action can be unbounded (each evaluator function can have unlimited number of parameters not bounded by \( n_a \), since the same attribute at different times can appear as a parameter).

`ESTABLISH` cost is dominated by the cost of Get and Set operations on the data structures if the attribute to be established is either a Constructor Attribute, or an Evaluated Expiring attribute. The number of Gets and Sets is \( O(n_a) \) in the worst realistic case (if the evaluating function has \( O(n_a) \) parameters) each of which are \( O(\log(n)) \) in the size of the history. In the case of dormant attributes there are two possibilities:

- If the state from which the attribute is retrieved is recorded, the cost is dominated by the cost of retrieving previous object-states. **Note (5)**: Given a current object-state, the cost of finding the previous object-state from the data structures is \( O(1) \). However, \( O(n) \) "previous object states" may have to be traversed.

- If the recorded history has to be parsed in order to derive the previous attribute value, the cost of `PARSE_RECORDED_HISTORY` \( O(n^3) \) (proposition 6.5) dominates the cost of `ESTABLISH`.

In many cases, (we anticipate that) the values of attributes in a newly created object-state will be calculated from neighboring states (such as the immediately preceding state).

- In this case, the cost of Gets \( O(\log(n)) \) for each of \( O(n_a) \) function parameters) dominates the cost, since the cost of evaluating each function itself is minimal (if actions are simple additions, concatenations, propagations etc.). Hence, the complexity of the actions is bounded by \( O(\log(n)) \) in most cases.

Given these figure, the overall cost of executing a reaction triggered by an update to an individual attribute may be given by the following expression:

\[
O(\log(n)) + O(C) + O(\log(n_a)) + O(C^*R) + O(C^*R)^*\log(C^*R) + O(n_a)^*O(ESTABLISH)
\]

where the first underlined term refers to the cost of getting and setting the updated attributes, the second represents the costs of finding out the current class memberships, the third represents the costs of finding the incident reactions and sorting them, and the last re-
represents the cost of evaluating $O(n_a)$ attributes. This expression yields our proposition above, under the constraints imposed. For example, if parsing is needed, the cost of parsing dominates the cost. If $n_a = O(\log(n))$ (unlikely), that gives a worst case realistic cost of $O(n^3 \log(n))$. In the usual worst case, where $v = O(n), n_a = O(1)$, and $O(ESTABLISH) = O(\log(n))$, we have a complexity of $O(\log(n))$. If $C = O(n)$, but $R = O(1)$, the complexity is $O(n^4 \log(n))$. However, the fact that the complexity can be unbounded or uncomputable must be acknowledged.

6.8.4 Effecting Inference: The ATN Definition as a Recognizer

If the entire derivation and recorded history are explicitly stored, the grammar-based algorithms may be expendable in favor of extensive indices on historical data. This option is not realistic. Therefore, many operations (and e.g. the DERIVE function in section 6.8.3) need to parse a succinct recorded history in order to recreate its derivation history, for purposes such as calculating attribute values which are dependent on previous states, matching histories for ON_HISTORY selection conditions, recreating and/or restoring previous states of recorded history, and for goal-driven operations. In such cases, the grammar is used bottom-up in order to infer implicit information in a recorded history.

Standard algorithms do exist for ATN-based systems. However, for reasons mentioned at the start of section 6.8, they are not suitable for dynamic and adaptive information systems. Since ATNs are CFGs with attached attributes, well-known context-free parsing algorithms can be augmented with attribute evaluation facilities. The most general top-down algorithm which may be adapted to this purpose is Earley's algorithm [79], and there exist more specific and efficient bottom-up algorithms for this purpose. However, we adopt the algorithm due to Cocke, Younger and Kasami (CYK) [173], for the following reasons:

- The CYK algorithm works for ambiguous and non-ambiguous grammars and it can be adapted for deterministic or non-deterministic use. These being desirable characteristics of dynamic domains, and CFG parsing being inherently $O(n^3)$ in the presence of these features, more efficient algorithms for restricted types of grammars are of limited value to dynamic and adaptive information systems.
- the CYK algorithm is easy to analyze and augment with semantic actions at different steps, and the results extend to more efficient but complex algorithms.
- the CYK algorithm is simple to specify and implement, since it involves little book-keeping and pre-compilations by the parser. It is thus a good candidate in situations where the dynamic definition changes frequently, and for prototypes.
Since most calls to parse a recorded history are made with respect to the most recent class membership(s), i.e. right to left, we adapt the CYK algorithm for right to left incremental scan. Since we also allow a limited form of length reduction (monotonic local) in our framework, the algorithm has been adapted for this purpose as well. Variables and terminal symbols being indistinguishable when a grammar is used for dynamic definitions, the two have been equated in our algorithm, although classes which may not be further enhanced are designated "terminal" classes. In addition to the normal parsing process, the attributes attached to the grammar symbols must be evaluated as the parsing proceeds, or after the parse tree has been created, as per the needs of a specific operation or the conditions on the attributes attached to the grammar symbols. The length-increasing rules of the grammar are assumed to be in CNF (Chomsky Normal Form) which describes all sentential forms. As noted earlier, terminal and non-terminal symbols are not distinguished. The length-reducing rules may be of the form $A_1..A_n \rightarrow A$. We recall from section 6.6.2 that $A$ is unique and does not occur in length-increasing rules.

```
reentrant PARSE_RECORDED_HISTORY(o)
1. i <- 1;
2. STATE[1,1] <- c \in \text{Collections}(o) /* assume a singleton class */
3. If (Find_CNFP_production(S->c)) \land (EVAL(S)=OK) \land (EVAL(c)=OK) \{ 
4. \quad STATE[1,1] <- STATE[1,1] \cup \{S\} 
5. \quad EXPAND(1);
6. \quad i := i+1;
7. \quad If (STATE[i, 1] = \emptyset) \{ STATE[i, 1] <- \text{Previous}(o,t,c) \} /*reentrance*/
8. \quad If (STATE[i, 1] = \emptyset) \{ return \}
9. \quad EXPAND(i);
10. \quad If ((Find_CNFP_production(S->c)) \land (EVAL(S)=OK) \land (EVAL(c)=OK)) \{ 
11. \quad STATE[i, 1] <- STATE[i, 1] \cup \{S\} 
12. \quad For (j=2 to i) do 
13. \quad \quad If (Find_CNFP_production(A->BC) \land 
14. \quad \quad (B \in STATE[(i-j+k+1),(j-k)]) \land 
15. \quad \quad (C \in STATE[(i-j+1), k]) \land 
16. \quad \quad (EVAL(A)=OK) \land (EVAL(B)=OK) \land (EVAL(C)=OK)) \{ 
17. \quad \quad STATE[(i-j+1),j] <- STATE[(i-j+1),j] \cup \{A\}; /*reentrance*/
18. \quad \quad If ((Find_CNFP_production(S->A)) \land (EVAL(S)=OK) \land 
19. \quad \quad \quad EVAL(A)=OK)) \{ 
20. \quad \quad \quad STATE[(i-j+1),j]<-STATE[(i-j+1),j]\cup\{S\};/*reentrance*/
21. \quad \quad \quad 
\}
\}
\}
```
EXPAND(i)
1. While \( \text{Find-CNF-Production}(A_1 \ldots A_n \rightarrow \text{STATE}[i, 1]) \) do 
   2. \( \text{Eval-and-Insert} \), \( \text{STATE}[i, 1] \leftarrow A_n \); \( \text{STATE}[i+n-1, 1] \leftarrow A_1 \) \)

After EXPAND, the row \( \text{STATE}[i,1] \) contains the uncompressed history generated by the length-increasing rules. We omit "Eval_and_Insert" since, as the name implies, it evaluates the attributes of, and inserts the states corresponding to LHS of the length reducing rules (section 6.6.2) starting at the in the i'th position in the list, while the occupants of the replaced positions are shifted to higher indices in the list. The function Previous refers to the \( \eta \) (time1, Class) Object function discussed in the next chapter. PARSE_RECORDED_HISTORY implements the core of the functionality provided by functions such as DERIVE and the SELECT operations where evaluation of an attribute requires parsing of the recorded history. Though it is based on the CYK algorithm but represents an original contribution in the following ways:

- It parses the history from right to left, as required when histories are used to find attributes of previous states, as against the left to right parse used in formal languages. It also uses EVAL to evaluate inherited and synthesized attributes. We omit EVAL from discussion because it is simply the calculation of a set of registers.
- It incrementally finds previous states, instead of assuming that the entire history is available at the start. It is also reentrant, to allow the parsing process to be interrupted whenever a new state in the derivation history is recovered. This enables the caller to examine the state, and resume execution at the point of interruption.
- It allows (through EXPAND) a limited form of length-contracting productions that enable histories to be summarized to be used in the parsing process of the history.

The analysis of correctness and performance of the algorithm follow in later sections.

Example 6.7  The use of a dynamic grammar as a recognizer is illustrated by extending our running example in order to exercise the different types of grammar rules.

Assume that a sales order (SO) may evolves during its existence through states In_prep, In_Accts, etc. as shown. In_Accts and In_Mats may be collapsed to In_Proc (appendix E). CNF rules expressing these relationships may include:

\( \text{SO } \cdashrightarrow \text{In Prep}; \text{ In Prep } \rightarrow \text{in prep.In Acct}; \)
A possible recorded history is in_pre.in_proc.Sls_Rec. If \( n_{Sls\_Rec}:\text{Category}(T_0) \), a constructor defined as "\( n_{SO}:\text{Category}(T_0) \)" is queried, the recorded history is parsed to derive a state where \( n_{SO}:\text{Category}(T_0) \) is defined. (Note: for simplicity, we assume that nothing precedes or follows SO, and that \( n_{SO}:\text{Category}(T_0) \) is somehow defined for SO’s. The figures below show successive views of the STATE table (i increases rightwards from 1, j increases upwards from 1):

```
Sls_Rec  // last class of object O
Sls_Rec.in_proc  // previous class of O
Sls_Rec.in_mat.in_acct  // expanding A_1..A_n -> A rules
In_Mat
Sls_Rec.in_mat.in_acct  // expanding A->BC rules
In_Acct
In_Mat
Sls_Rec.in_mat.in_acct  // expanding A -> BC rules
In_Acct
In_Mat
Sls_Rec.in_mat.in_acct.in_prep  // expanding A->BC rules
In_Prep
In_Acct
In_Mat
Sls_Rec.in_mat.in_acct.in_prep  // expanding A->BC rules
SO,In_Prep
In_Acct
In_Mat
Sls_Rec.in_mat.in_acct.in_prep  // expanding A->a rules
```

From SO, as derived in STATE[1, 4], the desired attribute value may be derived.
6.8.4.1 Correctness of Recognizer Algorithm

The correctness of the algorithm above is based on the proof of correctness of the original CYK algorithm. The proof itself is not innovative but justifies our claim for adequacy of our framework for reasoning with parsable histories defined by grammars.

- We omit details about allocation of memory when new cells of the STATE table are created or new elements assigned to existing cells.
- We illustrate the case for a single class membership. This is easily extended to multiple class memberships by modifying the STATE data structure.

The key invariant in establishing the correctness of PARSE_REC_HISTORY is: $\text{STATE}[i,j], j>1$ contains symbol A if and only if $A \rightarrow \star \text{STATE}[i+j-1, 1].\text{STATE}[i+j-2, 1]...\text{STATE}[i, 1]$. The notation $\rightarrow \star$ indicates grammatical derivability in zero or more steps by expanding grammar productions. If $\text{STATE}[1, n]$ contains the starting symbol $S$, where $n$ is the uncompressed length of the history, then the string $\text{STATE}[n, 1]...\text{STATE}[1, 1]$ (i.e. the uncompressed recorded history itself) is obviously derivable from $S$, which is the intent of the algorithm. The proof uses strong induction on the variable $j$ above.

1. (Base step) For $j=1$, the invariant implies: $\text{STATE}[i,1]$ contains A iff $A \rightarrow \star \text{STATE}[i,1]$. The proof of this step is as follows: The content of $\text{STATE}[i, 1]$ may be obtained from one of three sources:

   a. from step 2 or 7 of PARSE_REC_HISTORY, where $\text{STATE}[i,1]$ is directly retrieved from the recorded history. In this case, trivially, $\text{STATE}[i,1] \rightarrow \star \text{STATE}[i,1]$

   b. from step 4 or 10 of PARSE_REC_HISTORY, where $\text{STATE}[i,1] \leftarrow S$ if $c \in \text{STATE}[i,1]$ and $S \rightarrow c$. In this case, $\text{STATE}[i,1] \rightarrow \star \text{STATE}[i,1]$ (i.e. derives in a single step).

   c. If the symbol in $\text{STATE}[i,1]$ was inserted into $\text{STATE}$ by EXPAND, then trivially $A \rightarrow \star \text{STATE}[i,1]$, because EXPAND only replaces state-symbol $S$ retrieved from compressed history by a set of symbols in the original uncompressed history which $S$ replaced (discussed later).

2. (Inductive hypothesis) Assume that the invariant holds for all $i,j$ for $1 \leq j < j' \leq n$, where $n$ is the length of the uncompressed history.

3. (Inductive Step) Consider the elements in $\text{STATE}[i, j']$. Firstly note that by line 18 of PARSE_REC_HISTORY, and the fact that $i \leq n$, the length of the uncompressed history, $\text{STATE}[i,j]$ is defined only for $i, j$ such that $(i+j) \leq n+1$. $\text{STATE}[i, j']$ is modified in two ways:

   a. by step 20 of PARSE_REC_HISTORY: in this case, symbol $A \in \text{STATE}[i, j']$ if $c \in \text{STATE}[i, j']$, and $A \rightarrow c$ is a production in the grammar. To prove $A \in \text{STATE}[i,j']$ iff $A \rightarrow \star$...
CHAPTER 6: D'FOUNDATION - A COMPUTATIONAL FRAMEWORK

STATE[i+j'-1, 1]...STATE[i+j'-2, 1]...STATE[i, 1], it is enough to prove c ->* STATE[i+j'-1, 1].
STATE[i+j'-2, 1]...STATE[i, 1], since if c derives the string in n steps, A derives it in n+1 steps
deriving c in one step and then the string in n steps provided the grammar is cycle free (i.e. A
can not derive itself.). Cycle freedom is assured in the transformation of CFGs to CNF.

b. by step 17 of PARSE_REC_HISTORY: if A -> STATE[i,j'] by step 17, then we prove:

If cond.: A->* STATE[i+j'-1, 1]...STATE[i+j'-2, 1]...STATE[i, 1] => A ∈ STATE[i,j']. We first
note that since j'>1, the length |α| = STATE[i+j'-1, 1]...STATE[i, 1] > 1, i.e. that there are at
least 2 symbols in α, where A->*α. All rules applicable to STATE[i,j], j>1 are of the form A->a
or A->BC.

• Since |A| = 1 and |α| > 1, we can deduce that there is at least one rule of
the form A'->BC, such that A->*A' using length non-increasing rules like A->a, B->*β,
C->*χ, and α=β.χ. This follows from the concatenation properties of CFLs. (Since
A -> STATE[i,j'] by step 17 as per our assumption, we can assume A->0A', but this
is not a critical assumption.)

• Since α=STATE[i+j'-1, 1]...STATE[i, 1]=β.χ, there must be a k such that β=
STATE[i+j'-1, 1]...STATE[(i+k), 1] and χ=STATE[(i+k-1), 1]...STATE[i, 1], with the
restriction that 0 < k ≤ (j'-1) to keep the split point within the string boundaries.

• By our inductive hypothesis, then B ∈ STATE[(i+k),(j'-k)] and C ∈ STATE[i, k]

• Now step 17 of our algorithm hypothesis states that A ∈ STATE[i-j+1,j] if there is a produc-
tion A->BC, and a k such that B ∈ STATE[(i-j'+k+1),(j'-k)] and C ∈ STATE[(i-j'+1),
k]. Substituting the variables appropriately, we get A ∈ STATE[i,j'].

Only If cond.: A ∈ STATE[i,j'] => A->* STATE[i+j'-1, 1]...STATE[i+j'-2, 1]...STATE[i, 1].
The logic says that A ∈ STATE[i-j+1,j] if there is a production A->BC, and a k such that B ∈
STATE[(i-j'+k+1),(j'-k)] and C ∈ STATE[(i-j'+1), k]. If A -> STATE[i,j'] by step 17, then by
substituting the variables appropriately, we get:

• There must be a k such that: B ∈ STATE[(i+k),(j'-k)] and C ∈ STATE[i, k] and a pro-
duction A->BC.

• By our inductive hypothesis, if B and C belong to the locations as mentioned
above, then B->* STATE[(i+j'-1), 1]...STATE[(i+k), 1], and similarly C->*
STATE[(i+k-1), 1]...STATE[i, 1].

• Since A->BC, B->* STATE[(i+j'-1), 1]...STATE[(i+k), 1] and C->* STATE[(i+k-1), 1]
...STATE[i, 1], then by the properties of length-increasing CFG’s (length decreas-
ing rules do not apply to j>1), A->* STATE[(i+j'-1), 1]...STATE[(i+k), 1].STATE[(i+k-
1), 1]...STATE[i, 1]. But this is equivalent to A->* STATE[(i+j'-1), 1]...STATE[i, 1].
The proof shows that uncompressed histories are parsable by \textit{PARSE\_RECEIVED\_HISTORY}.

When a history $\alpha$ derived as $A \rightarrow^* \alpha$ is compressed to $B$ by length-reducing productions (section 6.6.2), the algorithm \textit{EXPAND} uncompresses $B$ incrementally to $A$, which is parsed correctly as shown above. The correctness of \textit{EXPAND} is informally argued (refer section 6.6.2):

- For a length-reducing production $A_1..A_n \rightarrow A$, the following restrictions apply:
  - $A$ is a unique \textbf{length-reducing} symbol: there is exactly one length reducing rule with $A$ on the rhs, and $A$ does not occur in the lhs or rhs of any length-increasing rule.
  - $A_i$, $i=1..n$ can be length-reducing or length-increasing symbol. If it is a length-increasing symbol, it has the same status as any other symbol generated by length increasing rules. If it is a length-reducing symbol, then as above, there is exactly one rule $B_1..B_m \rightarrow A_i$ with $A_i$ on the rhs, and \textit{EXPAND} recurses.

\textit{EXPAND} is called by \textit{PARSE\_RECEIVED\_HISTORY} when a length-reducing state $A$ is found in the recorded history. \textit{EXPAND} uses the unique rule with $A$ on the rhs, and replaces it with the corresponding lhs $A_1..A_n$. Thereafter, each $A_i$, $i=1..n$ is like any other state retrieved from the recorded history.

The arguments above prove the ability of the two algorithms mentioned above to parse both uncompressed and compressed histories.

### 6.8.4.2 Complexity of Recognizer Algorithm

The complexity of reasoning with ATNs as a recognizer is presented below as \textit{propositions} rather than theorem because the claim depends loosely on simplifying assumptions made in the following informal proof. Although inherent worst case undecidability or exponential complexity is faced, many real cases with acceptable cost exist. The complexity figures from this section are used in the previous section on generator algorithms.

\textit{Proposition 6.5} Computation of object history and inference of attributes is theoretically uncomputable. Under restrictions on the complexity of individual attribute calculation complexity and the number of attributes, parsing complexity is $O(n^3)$ in the history size. If the number of attributes is logarithmic in the size of the history, it is $O(n^3 \log(n))$.

\textit{Informal Proof:}

The proof of this complexity is based on the measure of \textit{PARSE\_RECORDED\_HISTORY} complexity being asymptotically $O(n^3)$ in the length of the uncompressed history, and the cost of uncompressing a compressed recorded history being $O(n)$. The overall complexity of
the algorithms given is expressed by the following formula:

\[ O(\text{Find}) + O(\text{Expand}) + O(\text{Length of history}) \times (O(\text{Find previous}) + O(\text{Expand}) + O(\text{Update State Table})) \]

\(O(\text{Find})\) is the cost of finding the initial class memberships of the object \(O\). The cost of finding the \(\text{Collections}(o)\) is \(O(\log(n))\) in the size of the history (as briefly discussed in chapter 7, operation \(\pi_{\text{ObjId,Time}} \Rightarrow \text{Collections}(O)\)). The cost of \(\text{Find \_CNF\_Production}\) is \(O(1)\) since this is a data dictionary function unrelated to the population of the database.

\(O(\text{Expand})\) is the cost of expanding a length_reduced state if it is retrieved from the recorded history. The cost of finding the unique production \(A_1 \ldots A_m \Rightarrow A\) if it exists is \(O(1)\), and the cost of inserting the lhs \(A_1 \ldots A_m\) starting at position \(\text{STATE}[i,1]\) is \(O(m)\). Since at most \(O(n)\) (\(n\) is the size of the uncompressed history) symbols may ever be inserted, the average cost of each call to EXPAND is \(O(1)\) if it is called to expand every retrieved state.

\(O(\text{Length of history})\) is our measured variable \(O(n)\). This multiplicative factor results because the variable \(i\) is incremented (step (6)) until no other symbol can be retrieved or expanded. This means \(i\) is incremented from 1 .. \(n\) in a loop from step (6) to step (21).

\(O(\text{Find previous})\) in the loop uses the \(\eta_{(\text{time1,Class}) \text{Object} \Rightarrow C_1 = \text{Collection( Object)time2}\) data structure operation, which is \(O(1)\) if \(\text{Class}\) is provided as an input parameter (chapter 7).

\(O(\text{Update State Table})\) executes two loops \(j=2\rightarrow i\), and within it, \(k=1\rightarrow j-1\). Since \(i\) is bounded by \(O(n)\), so is \(j=O(n)\) and \(k=O(n)\). In other words the loop executes \(O(n^2)\) times. Each time, it may \(\text{EVAL}\) \(O(R)\) registers belonging to classes A, B, and C, each in \(O(e_r)\) time.

This brings the complexity to: \(O(\log(n)) + O(1) + O(n) \times (O(1) + O(1) + O(n^2) \times O(R) \times O(e_r)) \). Theoretically, \(O(R)\) is unbounded because any number of attributes may be manifested by an object. Usually, it is bounded \(O(1)\). \(O(e_r)\), the cost of each evaluation can be uncomputable. If evaluations are simple additions, and checks are simple checks, \(O(1)\) cost is justified.

The overall complexity in thus \(O(n^3)\). If complexity of EVALs is assumed to be \(O(\log(n))\), then \(O(n^3 \log(n))\) may be assumed.

The CYK algorithm underlying \(\text{PARSE\_REC\_HISTORY}\) may be replaced by one such as Earley’s algorithm, which takes \(O(n^3)\) time for ambiguous grammars, \(O(n^2)\) time for unambiguous grammars and \(O(n)\) time for most. In such a case, the following statement (stated without proof; refer standard proofs for Earley’s algorithm) holds: The measure of parsing complexity of object history and inference of attributes is \(O(n^3)\) in the history size for ambiguous grammars, \(O(n^2)\) for unambiguous grammars, \(O(n)\) for most grammars. If the number of attributes is logarithmic in the history length.
6.8.5 Supporting Data Dictionary Functions

The execution of the algorithms given above requires support from the data dictionary for all functions given in italics. The data dictionary stewards the dynamic schema, independent of the current population of the information system. We enumerate the major data dictionary functions already referred to in the preceding sections and justify their complexity.

- **Find_Links**: Find reactions sourced and targeted at a node; **Find_ATN_for_Reac**: find the ATN links that interpret a given reaction - since the number of classes and nodes is small, complexity is independent of the database size and can be considered to be $O(1)$.

- **Find_CNF_Productions**: find $A \rightarrow BC$, $A_1..A_n \rightarrow A$, and $A \rightarrow a$ productions in CNF corresponding to an ATN - Conversion of the CFG denoted by the ATN to Chomsky Normal Form [79] is a pre-runtime activity. The runtime task of finding the productions is a $O(1)$ activity w.r.t. the set of objects, since the number of productions is smaller.

- **Pre_Conds, Post_Conds and Actions** associated with a given ATN link - If the number of nodes is $n_c$, the number of links may be $O(n_c^2)$ or more. However, $n_c$ is a small number and interpretations of dynamic behavior normally yield sparse grammars. Since pre- and post-conditions and actions associated with a link are known before runtime, we can assume $O(1)$ complexity.

- **Expiring_Attr(c), Eval_Attr(c), Defn_Attr(c)** ... attributes of classes - The definitional and factual attributes of classes are directly available from the definitions as exemplified in section 4.7. Their retrieval by the data dictionary is thus a trivial task of negligible computational complexity.

- **Generation(r), Concurrent_Branching(l), Token_Identification_Actions(r)**... - Find the classification of specific reactions and links - reaction types are specified directly in d' dialect(O). Whether or not actions like concurrent-branching or token-identification are associated with a link is implied by the standard interpretation of reactions in ATNs. This is determined before runtime; its retrieval is therefore $O(1)$, given that the number of links and reactions is much smaller than the number of objects $n$.

- **Define**: Define is used in EFFECT_REAC to create new registers. The registers to be created when a specific link is traversed are determined from the definitions of the source and target classes, and the set of factual attributes of the source class. This number is normally bounded $O(1)$ with respect to the number of classes $n_c$ or objects $n$. The cost of creating each individual register is $O(1)$.

- **Token_Id_Action(l) and Concurrent_Branch_Actions(l)** - These are precomputed according to definitions 6.3 and 6.4, and are not context-dependent. The data dictionary can retrieve the precomputed values in a single atomic $O(1)$ step.

- **Find_Path**: Reachability computation for pairs of nodes - Reachability between pairs of nodes is the transitive closure of the derivability relationships implicit in the ATN
grammar. Find_Path can be computed a priori using a standard breadth first prune and search based on pre-compiled knowledge regarding the reachability of states from each other and detection of circularity in the states visited. At each stage of the breadth-first expansion, the algorithm prunes the branches which revisit previously visited states, and those which reach states from which the target state is unreachable. After the pre-compilation, the paths to reach class b from class a can we retrieved in $O(1)$ time using an index on source and target.

Since data dictionary functions largely follow directly from the definitions or are analytical (pre-compiled), and concern small volumes of information, they are largely $O(1)$ and we do not elaborate further. Chapter 7 addresses the data structure support required by our algorithms, in terms of object-based, class-based and history-based queries and updates.

6.9 Chapter Summary

Chapter 6 contributes the computational framework underlying queries and updates that may be performed on information modeled using d'dialect, and constitutes a primary contribution of this thesis. We interpret the representing and reasoning with object histories through a well-known mechanism based on formal languages, and show the translation of the primitives of d’dialect to the terms of this framework.

Some operations on dynamic information are enumerated, and later categorized into two types depending on whether they use the grammar-based interpretation of dynamic behavior in a generative, or recognitive, manner. Thereafter, some core algorithms for each of these modes are given, and their dependencies on data structures is listed. We illustrate through this process the basis upon which grammar-based reasoning about complex time-indexed entities such as histories can be integrated into database query and update processing. The algorithms are derived from previous work in formal language processing, but are adapted to the task of reasoning with histories. The adaptation includes right-to-left reasoning, their relationship to operations on data structures, their limited support of length reduction during the evolution of histories, their ability to consider multiple concurrent object-states and their specialization with respect to attribute types such as dormant attributes, which are particular to dynamic information systems. Finally, some performance indications, and their adequacy in practical terms are discussed. The algorithms and their analysis are not our main focus; they are presented to support our claim of realizability for the expressive power offered by dynamic and adaptive information systems at an acceptable cost.
Chapter 7  Database Structures for Dynamic and Adaptive Information Systems

Inferences with dynamic object relationships rely upon data structures combining temporal query processing and multi-dimensional search, as the algorithms in the previous chapter illustrate. The crucial additional requirement imposed on data structures by adaptive information systems is the need to enable class and relationship definitions to change efficiently over time (chapter 8). We introduce data structures which enable reasoning with changing schemas, while supporting dynamic information systems in this chapter. Our proposals are inspired by work in computational geometry (see Appendix D) which addresses dynamic data structures and data structures for computing with intervals. We contribute some unique hybrids of these structures, and also the ideas on applying them to the area of dynamic and adaptive information systems in this thesis. We do not claim that our proposal is optimal, but we do claim that their performance proves the viability of our framework for practical systems. In summary, dynamic and adaptive information systems require that:

- Objects belonging to a particular class at a particular time, and the attributes defined on the objects, be reachable.
- Recorded historical information should be easily reachable for each attribute, as well as for the history of membership in classes for each object.
- Classes should be defined intensionally. When definitions change, the memberships and extensions of the new classes should be easily derivable.

Conceptual data structures are proposed to organize the information in dynamic and adaptive information systems as per the above requirements, without addressing file/page structures and algorithms (such as ISAM files or BLOBs) to implement this proposal.

7.1  Object and Historical Data Access Requirements

The kinds of operations required by dynamic and adaptive information systems follow.

7.1.1  Object-centered Access to Information

- $\sigma_{\text{Search Attributes}}(\text{Set}) \rightarrow \text{Objects}; \sigma_{\text{Object IDs}}(\text{Set}) \rightarrow \text{Objects}$
Given the identifying attributes or the identity of an object, the object itself should be accessible in $O(1)$ time if surrogate identities are supported by the technology, and $O(\log(n))$ time if a search based on inverted indices on the attribute values is used ($n$ is the number of objects). This compares with the efficiency of selection in a relational database. Such indices are well understood, and not discussed further.

\[ \pi_{\{\text{Attribute Name},\text{TimePoint}\}} \Rightarrow \text{Value/Definition(s)} \]

Assuming that given a time interval and other selection criteria, a set of objects has been identified, projection of the queried attribute values (including an attribute on the time dimension) themselves should be a constant time $O(1)$ operation for each attribute after the required intervals have been selected. The selection cost itself should compare with select and project operation in historical databases. However, the recorded history may yield only a definition of the value in terms of historical attribute values. The definition must then be evaluated using the algorithms given in section 6.8.

\[ \text{Assert } [\text{Attr Name}(\text{Obj Id}) \leftarrow \text{Value}]_{\text{Time Interval}} \Rightarrow [\text{Effect Class Changes}] \]

(see section 6.10.1, paragraph 7) The cost of storing in the data structures themselves must be comparable to $O(\log(h))$, as in historical databases. However, the action of updating an attribute in dynamic and adaptive information systems is inextricably associated with the implicit action of effecting the changes in categorization implied by the update. Since historical databases are incapable of inferring any changes to categorization, there is no benchmark for the implicit actions implied by an attribute update. The actual performance is limited by the algorithms described in section 6.10.

\[ \pi_{\text{Obj Id, Time}} \Rightarrow \text{Collections(O)} \]

This requirement for access function is important for dynamic and adaptive information systems since it is a prerequisite for other grammatical inferences using the ATN formalism (for example, in the preceding operation). It is absent in conventional historical databases because historical databases do not need to reason from objects to the classes they belong to at a given time, and because multiple class membership is not allowed in most existing historical database systems. Given an object id, a low processing time (in the size complexity of the object, i.e. number of attributes) is thus a requirement for this operation.

\[ \sigma_{\text{Time}} \text{ Set} \Rightarrow \text{Objects} \]

In the case of dynamic information systems, the selection criteria are extended to
include the extensions of specific classes at specific times. The requirement of locating objects that exist in a particular set at or during a particular time is similar to that placed in selection queries using inverted indices. The acceptable cost of executing such queries is also similar - logarithmic in the size of the time dimension.

The above operations and their acceptable costs associated with them extend the requirements on existing data structures. For example, they add the requirement of effecting changes in object categorization as a result of attribute update, and the ability to execute queries on the set of all objects, or on the extensions of specific classes at specific times. They also introduce new types of queries such as that for the class memberships of a given object at a given time.

In addition to these extensions, the data structures proposed in this thesis are designed to be particularly amenable to effecting changes to the definitions of classes cheaply, as indicated below.

7.1.2 Class-centered Access to Information

Most accesses in a database environment are oriented from the direction of the classes in terms of which existing objects is categorized. Typical queries in this category, and the acceptable cost for these queries are shown below.

- $\sigma_{(q,1)} \text{Class} \Rightarrow \text{Objects}$
  
  This expression refers to the request to fetch all members of a class at a given time $I$ with no further constraints. The cost of identifying all members of a class should be constant; that of fetching all members should not exceed $O(n)$, where $n$ is the size of the answer. This compares with the acceptable cost in existing databases.

- Select, Project, O-Join on sets of objects

  We do not address these operations specifically in this thesis, since the literature in this area contains references to many well-understood methods and indices to support these operations.

- $\sigma_{(Selection\ Criteria: Attr = Value, Interval)} \text{Class} \Rightarrow \text{Objects in aggregated super/sub-classes satisfying the criteria}$

  If the queried attribute is equated to an aggregate super- or sub- class of the class which is addressed in the query, the data structures must support static indices capable of fetching the appropriate object from the super/sub-class. Nested hierarchical indices [22] can be used...
to execute these queries efficiently, and are not discussed further in this thesis.

- \( \sigma_{\text{Selection Criteria: } \text{Attrib} = \text{Value, Interval}} \) \( \text{Class} \Rightarrow \text{Objects} \) in abstraction super/sub-classes satisfying the criteria

If the queried attribute is equated to an object in a generalization/abstraction super-class or subclass of the query class, the data structures associated with a dynamic information system should enable the retrieval of the objects in the related classes. As in the previous category, we propose the usability of nested-hierarchical indices in executing these queries, and omit further discussion in this thesis.

### 7.1.3 History-Centered Access to Information

- \( \eta_{(\text{time}_1, \text{Class})} \) \( \text{Object} \Rightarrow \text{C}_1 = \text{Collection(Objects)}_{\text{time}_2}, \text{time}_2 \text{ immediately precedes } \text{time}_1, \text{ and } x \in \text{C}_1 \text{ is legally connected to Class} \)

- \( \gamma_{(\text{time}_1, \text{Class})} \) \( \text{Object} \Rightarrow \text{C}_2 = \text{Collections(Objects)}_{\text{time}_2}, \text{time}_2 \text{ immediately follows } \text{time}_1, \text{ and } x \in \text{C}_2 \text{ is legally connected to Class} \)

Access to previous or future object-states in the recorded history is instrumental in the algorithms outlined in the previous chapter, in order to parse histories while inferring attribute values determined by dynamic relationships. Being primary to object evolution, the data structures must support this access at \( O(1) \) cost.

- \( \eta_{\alpha(\text{time}_1, \text{attribute})} \) \( \text{Object} \Rightarrow \text{Value-Interval} = I_{\text{Prec}}(\text{time}_1, \text{object}, \text{attribute}) \)

- \( \gamma_{\alpha(\text{time}_1, \text{attribute})} \) \( \text{Object} \Rightarrow \text{Value-Interval} = I_{\text{Foll}}(\text{time}_1, \text{object}, \text{attribute}) \)

These data structure functions return the immediately preceding or following value interval with respect to the query interval, where the specified attribute had a different value.

The value returned for the attribute may be a function of one of the following types

- \( = f(\text{preceding object-states}) \)

- \( = f(\text{derivative (subsequent) legally derived object-states}) \)

- \( = f(\text{attributes of generating object(s) of o}) \)

- \( = f(\text{attributes of objects generated from o}) \)

The pointers may refer to entities in the recorded or derivation history.

### 7.1.4 Requirements due to Space and Schema Adaptation

Apart from the bounds on time complexity for data access, the space complexity of the
data structures for temporal information must be bounded. Although up to $O(n^2)$ storage schemes are considered acceptable for certain static data (e.g., M:N relationships in relational systems), these are not built in but left at a user's discretion to use. Temporal data grows fast and a large number of secondary storage accesses is required to operate on it. We therefore consider a bound of $O(n)$ space complexity necessary for dynamic and adaptive information system data structures. As a counterpoint, we consider a scheme of $O(n \log(n))$ space which has advantages in terms of simplicity of the associated algorithms.

Adaptive information systems impose additional requirements on the data structures in terms of support for schema change. Ideally, we recognize the need to support three kinds of flexibilities in the associations among units of information:

- Association of attributes to objects.
- Association of objects to classes.
- Association of object-states to each other through time.

These flexibilities are needed because the definition of what constitutes a recognizable object or class may change over time. Adaptability requires the ability to change class definitions without incurring the performance penalty of modifying or reclassifying all existing objects. I.e., when class definitions change, the resulting class extensions should fall out easily from the data structures. Hence, the following class of functionality must be supported by the data structures underlying dynamic and adaptive information systems.

- $T_{\text{Union, Intersection and Redefinition}} \quad \text{Class(es)} \Rightarrow \text{Class(es)}$

The primary differentiator in our proposed data structures must be the ability to provide practical mechanisms for inexpensive schema redefinition to implement adaptive information systems, while preserving the cost constraints imposed by dynamic information systems on the other operations defined above.

The prime requirement on our proposed data structures is that the schema redefinition operations must have a cost proportional to the size of the schema, not the size of the extensional database conforming to the schema. Figure 7.1 illustrates the concept of change in class definitions in adaptive information systems. There are two attributes $\text{attr1}$ and $\text{attr2}$. The variation of the attribute values with time (assuming that they are real-valued attributes) for a particular object are shown by the dashed and the heavy lines, respectively. Class A and Class B are defined (for the sake of simplicity) as polygons extending through
time. The locus of the point \((attr1, attr2)\) through time passes through the polyhedrons labeled \(Class A\) and \(Class B\) (or both, or none) at different times. If the locus passes through a polyhedron, the object belongs to the corresponding class represented by the polyhedron, since the point satisfies the geometric constraints on all points in the interior of the polygon.

Class \(A\) and class \(B\) may be in the schema of a single observer at different times, or be in the schema of different users. The corresponding polygon shapes could change over time if the conception of the classes change. Regardless, individual object attributes or pointers to objects should not need to be manipulated in response to these changes, i.e. the locus of the points corresponding to the attribute values over time do not change as a result of the changes in class definitions. In contrast to this picture, we have the situation in relational database systems, where the introduction of a new classification implies the creation of new relations and the physical migration of every entity to the new relation set.

7.2 Introducing Dynamic/Adaptive Information Structures

The temporal information to be represented consists of time intervals (line segments in the temporal space) associated with attribute-states and object-states. The indexing structure imposed on this collection of line segments should be modifiable dynamically. Such conceptual views have been addressed in the area of computational geometry, but largely ignored in favor of database oriented file structures in the field of historical databases. We propose the realization of the dynamic and adaptive database functionality through hybrid data struc-
tures based on computational geometry techniques amenable to set-oriented access in databases. Our proposed data structures are founded on interval and segment tree hybrids with descriptors, with an important simplifying assumption - that the locus of an attribute's value over time traces disjoint straight-line segments in the two dimensions. This is the case for most non-numerical attributes and a significant number of numeric attributes (figure 7.2).

![Figure 7.2](image)

**FIGURE 7.2** Practical Evolution of Attribute Values through Time

To limit our problem, we do not allow attribute values to be defined by non-segment-wise functions over time, as in figure 7.3.

![Figure 7.3](image)

**FIGURE 7.3** Restrictions on Attribute Value Evolution through Time

This restriction is justified for most applications, though there exist applications which do not adhere to these restrictions. Area-coverage based data structures support searches over such segments (see for instance [109] and [167]), and are outside the scope of this thesis.

The basic temporal entities to be represented and searched are:

- **Attribute-Value (Time) Intervals**: (AV Interval) which are the time intervals associated with value intervals (definition 3.11) when an object manifests a particular value for a particular attribute. AV-intervals are used in value-based searches of sets of objects, as well as for reasoning with attributes on a temporal basis. AV-intervals are stored in the O-structure.
7.3 O-Structure: Framework for Dynamic Attribute Information

The record of an individual object's behavior over time is recorded as a recorded history in the O-structure. The O-structure for each object contains:

1. **Object Identity (OID)**, which is a Skolem function of some attributes, and may be abbreviated by a surrogate. A surrogate acts as an unambiguous address in object-oriented or network databases, and enables random access based on identity.

2. **AV-Interval List (AVInt List)** For each attribute associated with an object, we link the value intervals (AV-Intervals) in temporal order.

   a1 -> <\[v_1, t_1,t_2\],<\[v_2, t_2,t_3\], ... ,<\[v_n, t_{now-1}, t_{now}\]>

Each AVInt List(Object, Attribute) therefore has the following attributes:

- **AV_Ident** which uniquely identifies the AV-Int List
- **AV_List** which is the list of value intervals. Each AVInt has:
  - **Start-time** (\(\text{Start(Interval(AVInt))}\)) included in the interval
  - **End-Time** (\(\text{End(Interval(AVInt))}\)) not included in the interval
  - **Value** (\(\text{Value(Interval(AVInt))}\)) in one of three forms:
    - **Simple value**: Indices defined on such attributes refer to the native class.
    - **Constructor Function** of the form \(<\text{func, av}_1, \text{av}_2 ... \text{av}_n>\): func points to a function, and \(\text{av}_i\) s point to other AVInts representing function arguments.
    - **Reference pointer** to a historical object-state or another class: Indices defined on such attributes are historical or nested-hierarchical indices, respectively.
  - **AV Previous**, a pointer to the previous AV-Int in the AVInt List. \(\varphi\) for the first AVInt in the list.
  - **AV Next**, a pointer to the next AVInt in the AVInt List. \(\varphi\) for the last AVInt.

The structure above allows AVInts to be non-contiguous if there are intervals during which an attribute associated with an object has no defined value.
3. **AV-Interval Tree (AVInt Tree)**: a temporal interval tree index on the AV Ints. The main advantages of using an interval tree are:
   - \( O(n) \) storage w.r.t. the number of AV-Intervals
   - Synchronization of split points with the C-Structure (see section 7.5.2).

### 7.3.1 AVInt Tree: An Index on AV Intervals

The AV-Interval tree is an interval tree which associates a set of AV-Intervals with each node of the tree. The nodes (also referred to as split points) are labelled by time points as in figure 7.4. In the AVInt Tree of figure 7.4, the node 'e' stores all segments between \( 0(b,a) \). In general, the AVInts associated with node 'n' are those which are covered by the interval \((\text{leftparent}(n), \text{rightparent}(n))\) and which cover 'n'. Here,

- \( \text{leftparent}(n) \) = the node farthest from the root, of which either n or an ancestor of n is a right child.
- \( \text{rightparent}(n) \) = the node farthest from the root, of which either n or an ancestor of n is the left child.

**FIGURE 7.4** AV Int Tree

Each node 'n' has the following structure:

- **Endpt_Tree(n)**: A 2-dimensional tree for storing the start and end points of the set of AV-Intervals associated with n.
- **Descriptor(O)**: Attribute Value Descriptor (Data Descriptor or Hash Value), a succinct representation of object O's attribute values at the time point n (section 7.4).
- **CM_Pointer(O)**: Pointer to a CM-Interval for O in the C-Structure (see section 7.5).
- **Next_CMValue(O)**: Pointer to the next AVInt Tree node which points to a distinct CM-Interval for the object O.

The CM_Pointer(O) and Descriptor(O) attributes associated with each node of the...
AVInt Tree (of which there is one per object) are used to reason from object-states to the class membership(s) of the object at that specific time. The C-structure (section 7.5) provides a means to partition the non-temporal attribute space into regions that can be dynamically collected into classes as class definitions change. These attributes provide a first-level index into the region of the non-temporal attribute space where the object \(O\) resides at a specific time.

As data structures, the main innovation over the standard interval tree consists of:

- The use of the 2-dimensional tree secondary structure to store the AV-Intervals endpoints associated with each node.
- The use of descriptors and CM-structure pointers at each node, as a means to summarize the attribute values in the time intervals covering that node.

The effect of these structures on efficiency of computation is discussed in section 7.6.

### 7.3.1.1 Data Structure Alternatives

Alternatively, O-Structures can use a segment tree instead of an interval tree to index AVInts. AV-Segment storage is better in that segment trees are easier to compress, archive and balance. The main disadvantage is that segment trees consume \(O(n \log(n))\) space instead of the \(O(n)\) space of the interval tree. Segment trees and interval trees can not be differentiated on the basis of height, since this depends on the distribution of interval data. On the basis of these advantages, we use the interval tree as the primary data structure in this thesis.

### 7.4 Summary Descriptors of Object States

The O-structure aids in locating and reasoning with attributes at specific times, and in correlating temporally co-incident attributes for a given object. In many operations, there is a need to find and reason with the class which an object belongs to at a specific time. This is obtained from the C-structure, which partitions the non-temporal attribute space into closed volumes denoting classes. The position of an object within these volumes (classes) at the time point denoted by a node of the AVInt Tree is approximated by a coarse dynamic summary descriptor (hash value) attached to that node of the AVInt Tree for the object.

#### 7.4.1 Summary Descriptor Function

The summary data descriptor attached to node \(n\) of the AVInt Tree is an extensible hash value based on the attribute values of the AV-Intervals which cover \(n\). The function is:

\[
\text{Descriptor}(O)(a_1, \ldots, a_n) \mid_t = x ; \text{ where } a_i, i=1,n \text{ are the attribute values of object } O \text{ at time } t
\]
t. Descriptor(0) is a function but is neither injective nor surjective. It is:

- Coarse: it maps attribute values to hash values N:1; every attribute value update does not require an update of the descriptor for the relevant node (example 7.1).
- Not directly related to the definitions of classes at any particular point in time.

Thus, the summary description of an object-state at a specific time point

- changes only if the attribute values at that point change macroscopically,
- maintains its relative position on the 1-dimensional index of the attribute space even when class definitions based on that index change,
- adapts to new class definitions gradually through extensible hashing techniques.

**7.4.1.1 Extensible Hashing Scheme**

In order to have the above qualities, the summary descriptor has these characteristics:

- each attribute is hashed to an independent set of bits. Complex attributes may be reduced to one-dimensional attributes through transforms, e.g. z-transform and Gray transform. (Transforms are beyond the scope of this thesis.)
- "undefined" (φ) hashes to a distinguished value.

The data descriptor is meant to act as a coarse index into the C-structure. Thus, the number of distinct values which an attribute may hash into (Hence, the number of bits per attribute in the data descriptor) is not directly related to the size of the attribute range. Instead, since the data descriptor uses an extensible hashing scheme (see [127], [24], [26] and refs. for a review of hashing schemes), it is related to the number of classes which are distinguished on the basis of the attribute. Hence, a reasonable estimate of size is as follows:

- The number of bits in the descriptor for each attribute value is $O(\log(n_c))$, where $n_c$ is the number of defined classes.
- The size of the data descriptor is $O(n_a \log(n_c))$, where $n_a$ is the number of defined attributes and $n_c$ is the number of defined classes.

This size is acceptable for indexing purposes since there is one descriptor associated with each node in an object's AVInt Tree, with the following important caveat. We assume that the range of each attribute is subdivided in a simple way, so that the hashing function associated with it is simple as well (Ideally, the hashing function should have constant time complexity). If the range space of the attributes are arbitrarily divided among the different classes, then the complexity of calculating the descriptor when an attribute value changes (indexing overhead) can be prohibitive. In the worst case, either of the following is possible:
The $O(\log(n_c))$ space limit on the number of bits per attribute is not valid; or

- The time complexity of calculating the descriptor become $O(r_a)$, where $r_a$ is the size of the attribute range.

In adaptive information systems, the definitions of classes can change over time. To ensure that the descriptor remains relevant as a means to classify objects with respect to the current set of classes, an extensible hashing scheme is used. Extensible hashing implies:

- The number of bits encoding a particular attribute in the descriptor is dynamically modified depending on the attribute’s value in distinguishing among classes. I.e. if an attribute range of $a=[5,10]$ distinguishes a class $A$ from a distinct class $B$ which uses a range of $a=[11,15]$, then a single bit for $a$ in the descriptor would distinguish $A$ from $B$. If class definition changes later cause the ranges $a=[5,7]$, $a=[8,10]$ and $a=[11,15]$ to distinguish between three classes $A$, $B$ and $C$, an extensible hashing scheme would dynamically modify the number of bits for attribute $a$ from 1 to 2, since 2 bits are needed to distinguish the three ranges.

- Descriptor bit-strings dedicated to different attributes can be defined to be split-only (or merge-only) in order for needed distinctions to appear (and irrelevant distinctions to disappear, respectively) as class definitions change.

**Example 7.1** Figure 7.5 shows a summary descriptor based on an extensible hashing scheme, in the context of a change in the schema.

![Diagram of attribute encoding](image)

**FIGURE 7.5** Evolution of a Summary Descriptor over time

At TIME1, attribute $a$ is encoded by a single bit in the descriptor, and the descriptor function is $|a/11|$ which evaluates to 0 if $a = 8$ as shown. At TIME2, when the definitions of the classes have changed, $a$ may be encoded using 2 bits, and the descriptor function may be $|a/8| + |a/11|$, which evaluates to 1 if $a = 8$. The reason for the change may be as given above: the need to use attribute $a$ to distinguish classes $A$, $B$ and $C$, as again $A$ and $B$ at TIME1
Class B may be defined by a template which constrains the first 2 bits of the descriptor such that the first bit must be reset and the second bit set, all subsequent bits being "don't care".

Note that when attribute $a$ is updated during TIME 2 to $a=9$, the descriptor need not be updated, while it must be updated if $a=11$.

The extensible descriptor is used as a quick index into the (set of) class(es) which an object may be a member of at a given time as follows.

- Each class has a descriptor template using 0, 1 and X (don't care) bits, which is used to narrow down the set of possible class memberships of the object.
- When an attribute $a$ is updated from $X$ to $Y$ in $[t_1, t_2)$, the following occurs:
  - The descriptor is recalculated using the hashing function associated with $a$.
  - The new descriptor defines a point in the C-Structure's space. The C-structure (section 7.5.2) is used to check if the point falls within a valid class.
  - The knowledge of the $\text{Classes}(o)\mid_{[t_1, t_2)}$ and the legal reactions is used to validate the new affiliation(s) of $o$.

When an attribute is updated, the CM-interval associated with the relevant node of the AVInt Tree may also change leading to index update. This is addressed in section 7.5, but we mention a bound on the complexity of that operation through the following:

**Lemma 1** Each node $n$ of the AVInt tree has $\leq 1$ CM-Interval associated with it.

**Reasoning:** A node corresponds to a timepoint $t$ where any attribute $a$ can have at most one value. Since each attribute hashes to a distinct set of bits in the descriptor, the descriptor at each node has at most one value. Hence, the maximum number of CM-intervals is bounded above by 1. Given two nodes $n$ and $m = \text{leftparent}(n)$ (or rightparent($n$)), it is possible that $\text{Descriptor}(m) = \text{Descriptor}(n)$ even if for all attributes $a_i$, $a_i \mid_m =/\neq a_i \mid_n$ (descriptor function is coarse). In this case, the descriptor and CM-Interval are associated with the higher node of the AV-Int tree only, excluding the lower node.

### 7.5 C-Structure: A Framework for Dynamic Classes

The O-structure provides a temporal index on the attribute values associated with objects. These objects are aggregated into classes. An important premise of adaptive information systems is that class definitions change over time, as users' priorities change (see chapter 8). A group of objects which was recognized as a distinguishable category at one time may cease to be relevant at a future time, and other classes may crystallize instead. Unlike con-
Conventional databases, the structures used to organize the objects must be flexible in order to enable restructuring at low cost. The C-Structure is the means towards this flexibility. While the O-structure serves object-oriented operations, the C-structure serves class-oriented operations, where a group of objects belonging to a class is accessed, selected from and manipulated at a time \( t \). Analogous to the O-structure, the C-structure organizes CM-Intervals.

### 7.5.0.1 Limitations of the C-Structure

The C-structure is limited with respect to the range of possible changes envisioned for adaptive information systems in chapter 8. Neither the C-structure nor the O-structure support the reformulation of the notion of objects themselves (i.e. what constitutes a recognizable object?). Data structures to support this functionality remain a topic for future research.

In the most general case, a class may be defined as an extensional collection of objects with no conjunctive rules for defining membership (disjunctive definitions always exist). In this case, both the update of descriptors when an attribute value is modified and domain reorganization degenerate to \( \mathcal{O}(n) \) complexity in time, where \( n \) is the number of objects. In order for the C-structure to demonstrate tangible benefits, classes must be intensionally defined as per the following independence assumptions (which are the norm in practice).

### 7.5.0.2 Assumptions

- Number of classes \( n_c \) is independent of the number of distinct attributes \( n_a \) and in general, \( n_c \ll n_a \)
- Number of attributes \( n_a \) is independent of the number of distinguishable objects \( n_o \), and in general, \( n_a \ll n_o \)
- Apart from the limitations on power of the logic for defining classes, the length (number of propositions) of a class definition \( l_c \) is independent of the number of objects in the class, and no worse than linear with a small constant in \( n_a \).

This suggests (for the data structures, not for adaptive information systems in general) that classes are defined by disjunctive and conjunctive formulae on predicates asserting membership in contiguous subranges of attribute domains. Attributes are usable as definitional attributes, or a search keys for content-based access, or both. The C-structure enables this dual use by combining an attribute search space with a temporal search space.

### 7.5.1 CM Intervals and Structures

Intervals during which an object has attribute values which are macroscopically indis-
tistinguishable (with respect to their summary descriptors) are called CM intervals (CM-Ints or Class Membership Intervals). When the schema (set of classes) is stable, a specific CMInt could correspond to an interval during which an object is a member of a given class. However, in general, this is not true. Immediately after a schema redefinition, a CM-Interval may span more than one object-state. At any time, more than one contiguous CM-Interval associated with an object could correspond to an object's membership in a single class.

**Example 7.2** The figure 7.6 shows CM-Intervals in relation to the attribute values of a given object. The thin lines denote AV-Ints corresponding to the named attributes on the timeline below. The thick lines denote CM-Ints with the descriptor values labeled against them as ",# = X". Attribute value updates lead to new descriptor values (not always, since descriptors are coarse), and thus distinct CM-Ints. If two CM-Ints carry the same descriptor value, they are likely to denote intervals when the object belonged to the same set of classes.

![CM-Ints in relation to AV-Ints](image)

CM-Intervals have two aspects: an interval which defines the period of time during which the attribute values are substantially indistinguishable; and a set of non-temporal attributes defining a point in an indexed n-dimensional attribute range space during the CM-Interval. The latter index is more granular than the indexing of the attribute range space implied by the summary descriptor. Formally, each CMInt(Object) comprises the following:

- **CM_Node** which uniquely identifies the CMInt Tree node where it is located.
- **CM_Value(CM_Id)**, a *value interval*, which comprises an interval and a value:
  - **Start-time(Interval (CM_Id))** which is included in the interval
• **End-Time***(Interval(CM_Id)) which is not included in the interval

• **CM Descriptor** (CM_Id) which is the summary descriptor for the CMInt

• **CM Previous**, a pointer to the previous CMInt for the object, \( \varphi \) for the first CMInt.

• **CM Next**, a pointer to the next CMInt for the object, \( \varphi \) for the last CMInt.

• **FirstValue***(Attribute) for every attribute, a pointer to the first AV-Interval(Object, Attribute) which covers the CMInt.

• **LastValue***(Attribute) for every attribute, a pointer to the last AV-Interval (Object, Attribute) which covers the CMInt.

### 7.5.2 C-Structure

The C-Structure is a hybrid data structure with two parts capturing and indexing the temporal and non-temporal aspects, each optimized for different types of queries. The temporal structure (CMInt Tree) is optimized for interval coverage and overlap queries, and the attribute space (AR Tree) is optimized for range queries.

#### 7.5.2.1 CM Interval Tree

Temporal indexing is provided by a **CM-Interval Tree** (CMInt Tree), which is the primary data structure within the C-structure. It is a single interval tree storing CM-Intervals associated with all objects in the database. The split points associated with the nodes of the CMInt Tree correspond to those of the AVInt trees associated with each object, in order to easily relate AVInt Trees and CMInt Trees. Each CMInt Tree node \( n \) (\( n \) is a time point) has:

• **Endpt-Tree\( (n)\)** A balanced 2-dimensional tree of left and right endpoints of the CM-Intervals stored at \( n \).

• **Left-Child\( (n)\), Right_Child\( (n)\) and Parent\( (n)\)**: Pointers to children and parent nodes of the CMInt Tree.

• **TB-Region\( (n)\)**: Pointer to an Attribute Range Tree (**AR\( (n)\)** Tree), a secondary structure to index non-temporal attributes corresponding to the CMInts stored at \( n \).

Each node of the CMInt tree corresponds to a time-point \( N \) and stores a number \( m \) of CMInts, corresponding to \( m \) distinct objects whose attribute values are macroscopically indistinguishable at \( N \). In figure 7.7, \( CM(a,b) \) represents an interval covering time point \( n \), when object \( a \) has descriptor \( b \). Objects 1 and 4 may belong to the same class during CM Intervals \( CM(1,X) \) and \( CM(4,X) \) since they share the same descriptor value \( X \), while objects 2 and 3 may belong to distinct classes by virtue of having distinct descriptors \( Y \) and \( Z \).
By virtue of having different descriptors, objects 2 and 3 belong to distinct regions of the k-dimensional attribute range space, while objects 1 and 4 belong in the same region, which is distinct from the regions within which either of 2 or 3 fall. The regions of the attribute range space are indexed by the AR Tree described in the following section.

FIGURE 7.7  CM-Interval Example

7.5.2.2 AR Tree

The Attribute Range Tree (AR Tree) is the secondary component of the C-Structure. The AR(n)Tree is a k-dimensional R+ tree, if k is the number of attributes. The AR Tree is a union of AR(n) trees, each associated with a node (n) of the CMInt Tree. The figure 7.8 shows an AR-tree as a union of AR(1), AR(2), and AR(3) trees associated with the nodes 1, 2 and 3.

We define the parts of our secondary data structures as follows; complexity of certain operations on these structures is addressed in section 7.6.

Time-Bound Region each leaf of a AR(i) k-dimensional R+ Tree is called a Time Bound
Region (TBR). Each time-bound region contains a set of CMInts. Time bound regions are linked by a pointer **TIR-Ptr** to Time Independent Regions in an N:1 cardinality.

**Time-Independent Region** - each leaf of the AR k-dimensional R+ Tree is called a Time Independent Region (TIR). Each TIR has:

- **TBR_Ptrs()**: Set of pointers to TBRs whose union constitutes the TIR.
- **Class_Containers()**: Set of pointers to Class Containers or other TIRs. A single TIR may be contained in N class containers. Each CMInt in the TIR belongs in each class container referenced (directly or indirectly) by a Class_Container pointer.

TBRs and TIRs may be boxes of a standard size (similar hyper-cubes with distinct anchors). In planning applications, this is often justifiable, since classes are frequently distinguished by regular gradations of difference in some key attributes. If class definitions and granularity of the summary descriptors are based on regular gradations, the attribute space will be indexed regularly. Since class definitions and ad-hoc query conditions can not be restricted in general, this strong restriction is not valid in all cases, leading to limitations in the applicability of our data structures for general adaptive databases.

**Class Container** - A region on the k-dimensional attribute space which defines a class is called a class container. Equivalently, a collection of TIRs which define a class is labelled a class container. By implication, class containers are defined by a union of closed regions in the k-dimensional attribute space which constitute a user-distinguishable class at any time.

![Time-Independent Regions & Class Containers](image)

**FIGURE 7.9**

Each class container has:

- **TIR-Set()**: Since more than one TIR may be contained within a class container, the union of the object-states associated with the CMInts in the TIRs constitute the
CHAPTER 7. DATABASE STRUCTURES FOR DYNAMIC/ADAPTIVE INFO. SYSTEMS

extension of the class over all time. As mentioned earlier, the CMInts in the TIRs are indexed on the time dimension by the CMInt Tree.

- **Name()**: Class name

In figure 7.9, the TIRs in the attribute space fall within one or more class containers defined in the k-dimensional attribute space. Transitivity, so do the CMInts associated with each TIR. This may be implemented through M:N pointers as illustrated in figure 7.10, or through other alternative methods (we omit the alternatives in this thesis).

![Class Container Diagram](image)

**FIGURE 7.10** Physical Representation 1

When a class is redefined, incremental changes may be made to the containment relationships among TIRs. The *union-find* algorithm may then be used to improve access to set members over successive accesses to the class by reducing the length of the path traversed to reach the class contained over each successive access.

7.5.2.3 **Indices**

Conventional databases index each class on certain attributes, so that a specific value indexes onto a set of entities manifesting the given value for the attribute. In an object-oriented database, this capability is extended so that each class is indexed through attributes to more abstract classes and encapsulating classes as well. In the context of Time-independent regions (TIRs), the following indices may be defined on TIRs, as in [22] (see figure 7.11).

- **Nested-hierarchical indices**: If class B has an attribute X which is derived from class A, its encompassing class, then the leaves of the TIR trees at TIR3 and TIR4 have pointers to TIR1 and TIR2, where inverted indices to the objects in TIR3 and TIR4 are associated. A similar scheme for attributes derived from abstract classes may be defined with respect to TIR5 and TIR1/2.

These indices are defined on the CM-intervals in the TIRs. Since TIRs themselves are defined on the basis of a summary description of attributes, they themselves function as top-
level indices on the stored attributes. More granular inverted indices can be defined on appropriate TIRs (and created/deleted if classes are redefined) as requirements dictate.

In addition to nested-hierarchical indices, dynamic information systems need to search the extensions of temporally related classes based on the values of attributes associated with a target class. For instance, suppose a query demands all the members of a class A such that \( a(o \in A) = x \). Suppose moreover that A derives the attribute a from an attribute b associated with a class B, such that \( B \rightarrow^* A \), where \( \rightarrow \) represents any dynamic relationship (for simplicity, we may assume that \( \rightarrow \) is a single step relationship). This needs an index on class B based on the attribute b with members of A being its range. These are historical indices.

Historical indices are indices based on attribute values associated with CM-Ints contained in one TIR of class A, to CM-Ints in other TIRs associated with other classes related to A by some reaction. They are of three kinds as shown in figure 7.12. In the figure, attribute b is assumed to point directly to a generating object in class B, attribute a points to a more abstract object in class A, and attribute d to a subsequent object-state in class D.
- Transition_Scope Indices - indices from CM-Ints in a TIR associated with a class A to previous or subsequent states of the object, as per the recorded history. Indices of this kind may be defined transitively over multiple transitions.

- Derivation_Scope Indices - indices relating objects in one class A to objects in another class B such that B -->* A through expansion of subnetwork B "DURING" B’s validity interval. Derivation-scope indices can also be transitive.

- Generation_Scope Indices - indices based on the inverse of the G_Source relationships. Generation-scope indices relate the attributes of a generating class to that of a generated class if a direct generation relates them.

These indices are updated whenever a CMInt is added to or deleted from a TIR. The implementation of such historical indices in terms of data structures is similar to that of nested-hierarchical indices, and is beyond the scope of this thesis.

Together, the proposed hybrid data structures can be depicted as follows. The labels in black circles in figure 7.13 are used in explaining complexity in section 7.6. The shaded arrows denote the pointers among different data structures as explained earlier. The data structures themselves appear in the boxes, O-structure on the left, C-Structure on the right.
7.6 Complexity of the Dynamic Information Access

We discuss the operations impacted by the data structures above (referring to figure 7.13 in BOLD as required). We omit the analysis of operations that are internal to component structures such as interval trees and range trees, since these are derived from previous work (appendix D). Instead, we concentrate on the results due to hybridization of these structures.

The data structures proposed above consume $O(n)$ space if $n$ is the number of AV-Ints and CM-Ints in the database for all objects. This follows directly from the space complexity of interval trees, upon which our proposals are founded (appendix D). This space complexity is satisfactory, even in comparison with attribute-timestamped temporal database schemes, which have the best temporal database space complexity. In general, the time cost of data structure operations in this scheme is $O(\log(n))$ in the size of the search domain (be it intervals or attributes), owing to the fact that trees are used to index all structures. In this section, we examine critically each of the data structure operations in section 7.1 against the proposed data structures. We factor in assumptions such as those of sections 7.5 and 6.2.1 to derive the computational complexity due to data retrieval.

- $\pi\{\text{Attribute Name, TimePoint}\} \Rightarrow \text{Value/Definition(s)}$

To project the value of individual attributes at time $t$, we need the set of AVInt Tree nodes which may store AVInts covering $t$. If $t = N$ for some node $N$, then this set includes all nodes from the root of structure $I$ to $N$, inclusive. Otherwise, it includes all nodes on the path from the root to some leaf $i$ s.t. $N = \text{leftparent}(i)$ or $N = \text{rightparent}(i)$. In the worst case, this implies one node at each level of structure $I$ for each object, or $O(\log(h))$ nodes if $h$ is the number of nodes. At each such node $N$, $\text{Endpt\_Tree}(N)$ is searched for AVInts associated with $\text{Attribute\_Name}(o_i)$ for every object $o_i$ in the query target set. $\text{Endpoint\_Tree}(N)$ (a 2d tree) needs $O(\log(n))$ time to search if $n$ is the number of intervals stored at $N$. Under the assumptions of section 7.6.1, if $n$ is bounded by $C$, it implies a node-set of size $h = O(m/C)$, where $m$ is the size of the object's history in terms of number of AVInts. The combined search time is thus $O(\log(m/C)) \times O(\log(C))$, or $O(\log(m))$ for each object in the query set. $O(1)$ time is required to project (not calculate) values/definitions given the AVInts. This figure is inherently dependent on the size of the query target set, but can be insulated from the number of objects as a whole by associating different AVInt trees with each object, as we propose. It compares with search time in temporal databases. The complexity of calculating attribute values using constructor functions is outside the scope of the data structure (see chapter 6).
Finding memberships of an object starts by locating the lowest node \( L \) in the AVInt Tree (structure \( I \)) which stores AVInts covering the point \( \text{Time} \). We traverse \( I \) from \( L \) towards its root to find the first node \( F \) such that \( \text{Descriptor}(O) \mid F \neq \emptyset \). At worst, this search spans the height of the AVInt tree, or \( O(\log(h)) \). We access \( \text{CMPointer}(O) \mid F \) (B) to find \( X = \text{CMInt}(O) \mid \text{Time} \), and then \( \text{CMNode}(X) = N \), the node of the CM-Int tree \( II \) in \( O(1) \) time. \( \text{TBRegion}(N) \) (E) leads to the \( \text{AR}(N) \)-Tree \( A \) containing \( \text{descriptor}(O) \mid F \). \( \text{TIRPtr}(A) \) (F) leads to the time-independent region \( R \) which subsumes \( \text{TBR}(O) \mid F \). Following \( \text{ClassContainers}(R) \) (G), we access the class containers which contain \( R \) in \( O(1) \) time according to the assumptions in section 7.5.0.2. \( \text{Name}(C) \) for \( C \) in \( \text{ClassContainers}(R) \) constitutes the answer. The complexity of this operation is dominated by the cost of locating a node in the AVInt tree which has a descriptor. The remaining steps are \( O(1) \), including the last step, where the union-find algorithm for reorganizing subset inclusion over time bounds the cost of locating a class container given a TIR to the inverse of the Ackerman's function (appendix D).

To find the objects in Class during any interval universally overlapping \( I \), we locate the class container \( C \) such that \( \text{Name}(C) = \text{Class} \) in \( O(1) \) time, under the assumptions in 7.5.0.2. \( \text{TIRSet}(C) \) (G) point to the set of TIRs \( T \) contained in \( C \) (\( T < < h \)). We label as \( K \) the set of CM-Ints \( c \) such that \( c \in t \), where \( t \in T \). This set is the union of \( O(h) \) \( \text{TBR}(n)'s \), since each TIR points to \( O(h) \) \( \text{TBR}'s \) through \( \text{TBRPtr}(F) \). To evaluate the time dimension, search the CM-Int Tree \( II \) as follows:

- Locate the highest node \( n \) of \( II \) such that \( I \) covers \( n \) in \( O(\log(h)) \) time.
- Search \( \text{EndptTree}(p) \) for every ancestor node \( p \) of \( n \) in the CMInt tree for intervals overlapping \( I \), in \( O(\log(h)) \cdot O(C) \) time. Reasoning: intervals that cover \( I \) can not be stored in a descendant of \( n \), because occupants of descendants of \( n \) do not cover \( n \), and hence do not cover \( I \).

(\( \text{EndptTree}(p) \) for every child node \( p \) of \( n \), such that \( p > \text{Start}(I) \) and \( p < \text{End}(I) \) must be searched for intervals overlapped by \( I \) if the query is an existential overlap query, yielding \( O(h) \) Endpt-tree searches).

If the result is a set \( L \) of CM-Ints, then the query result is \( \text{Oid(FirstValue}(a))(K \cap L) \) following link (C). This function indicates finding the object id associated with the first AV-Int associated with any attribute of the object, which take \( O(1) \) time for a set of objects. The over-
all cost is dominated by the cost of the join, or $O(m)$, where $m$ is maximum(|$K_1$|, |$L_1$|). In most cases, this is dominated by $O(\log(h))$ cost of searching the CM-Int tree.

- $\eta_{(time_1, Class)}(Object) \Rightarrow C_1 = Collection(Object)_{time_2}$, time_2 precedes time_1, and $x \in C_1$ is legally related to Class

- $\gamma_{(time_1, Class)}(Object) \Rightarrow C_2 = Collections(Object)_{time_2}$, time_2 follows time_1, and $x \in C_2$ is legally related to Class

1. This operation first requires the establishment of the class memberships (in terms of the CMInt) of the query object according to $\pi_{Obj\_id, time_1} \Rightarrow Collections(O)_{time_1}$ at a cost of $O(\log(h))$. However, this operation is most often required in the context of a pre-determined CMInt. Hence in general, it is executable in $O(1)$. As per section 7.5.0.2, |Collections(O)| is a small number.

2. $CM\_Prev(X)$ (resp. $CM\_Next(X)$) (D) associated with $X=CM\_Int(O)_{time_1}$ (step 1) are followed to retrieve the previous (resp. following) CM-Int, $P$. $CM\_Node(P) = N$ of the CM-Int tree $II$ is found in $O(1)$ time. $TB\_Region(N)$ (E) gives us the AR(N)-Tree $A$ containing descriptor($O$)$_{time_2}$. $TIR\_Ptr(A)$ (F) leads to the time-independent region $R$ containing $TBR(O)_{time_2}$. $Class\_Containers(R)$ (G) points to the class containers containing $R$ in $O(1)$ time. Name($C$) for $C \in Class\_Containers(R)$ is the answer. If Name($C$) = Collections($O$)$_{time_1}$, merge the CM-Ints and repeat step 2.

The overall cost is max($m$, $n$), where $m$ is $O(\log(h))$ or $O(1)$ as stated in step 1, and $n$ is the number of repetitions of step 2. If many consecutive CM-Ints associated with $O$ refer to TIRs within the same class, 2 can be repeated $O(m)$ times in the size of the history in the worst case. However, (appendix D), after the first time, the cost reduces to $O(1)$ because we merge the intervals. The cost for this operation is therefore $O(1)$ for most cases.

- $\eta_{(time_1, attribute)}(Object) \Rightarrow Value\_Interval = I\_Prec(time_1, object, attribute)

- $\gamma_{(time_1, attribute)}(Object) \Rightarrow Value\_Interval = I\_Foll(time_1, object, attribute)

These functions require the identification of an AVInt corresponding to the value of the query attribute at time $time_1$. As per $\pi_{(Attribute\_Name, TimePoint)}$, this can take $O(\log(m))$ in the size of the object history. However, since they are invoked in the context of an identified AV-Int, they are most often executable in $O(1)$ cost. The AV_Previous and AV_Next pointers (A) associated with the identified AV-Int are followed at $O(1)$ cost, in order to complete this query, at a total cost of $O(1)$.

- Assert $[Attr\_Name(Obj\_Id) \leftarrow Value \mid Time\_Interval]$

Updating an attribute affects primarily the AVInt tree if class definitions don’t change,
but may cause recalculation of descriptor and CMInt values, as shown in the figure below. 

New AV-Interval

All nodes, intervals and descriptors marked by asterisk may be affected by the update of the attribute value during the new interval, where the existing values during the same period are indicated by thin lines. Locating the highest node H of the AV_Int Tree I covered by TimeInterval is $O(\log(h))$. Existing AV-Ints for Attr_Name(Obj_Id) covered by TimeInterval can be resident at $O(\log(h))$ nodes of the AV_Int Tree. All AV-Ints covered by TimeInterval are deleted in $O(\log(h))$ time and replaced by a new AV-Int at node H in $O(1)$ time, including realigning the pointers of the AV-Int list (A). In the worst case, $O(\log(h))$ Descriptor(O) values have to be calculated at each of the affected nodes, each in $O(1)$ time under the assumptions of section 7.4. Correspondingly, $O(\log(h))$ CM_Ints are created and inserted in the CM-Int tree II in the worst case. The cost of updating upto $O(\log(h))$ CM_Pointer(O) (B) in the O-Structure, and the cost of insertion of upto $O(\log(h))$ CMInts in the doubly linked list (D) in II is $O(\log(h))$ in the worst case. $O(\log(h))$ cost is inherent in this operation. Most updates span the interval I s.t. Start(I) = $t_{now}$ i.e starting at current time, where cost is $O(1)$.

- \(T_{\{\text{Union, Intersection and Redefinition}\}} \text{ Class(es)} \Rightarrow \text{Class(es)}\)

The proposed data structures are designed to support limited redefinition of classes via unions and intersections of regions in the attribute space, at the cost of higher conceptual complexity for other data operations. The TIRs (time independent regions) of the k-dimensional attribute space are the basis for schema reformulation. TIRs define coarse-grained regions of the non-temporal attribute space which may be combined into recognizable classes. (These regions can grow or shrink based on the redefinition of the extensible hashing function, see section 7.4.) Schema reformulation is done by updating the Class_Container[] pointers (G) associated with a TIR to point to new or different class-containers. For instance:

- If a new class C3 is defined so that the objects in the attribute region TIR1 (previously in C1) should belong to C3, then Class_Container(TIR1) $\leftarrow$ (Class_Container(TIR1) - C1) $\cup$ C3.

- If a new class C3 is defined so that the objects in the attribute region TIR1 (previously in C1) should belong to C3 as well as C1, then Class_Container(TIR1) $\leftarrow$ (Class_Container(TIR1) $\cup$ C3).
If TIR1 and TIR2 are combined due to loss of differentiating attributes:
- The extensible hashing is changed to drop the bits for the lost attributes.
- The AR(n)- and AR-Tree attribute dimensions are modified, and
- Class_CONTAINER(TIR2) <- TIR1 (the reference to TIR1 is replaced by the reference to the class itself during later accesses (appendix D).

The backward pointers TIR-SET[] and TBR-Ptrs[] are modified appropriately to maintain the reachability of the class extension from the class container. These operations cost $O(n_c)$ in the number of classes, which is independent of the number of objects, if in addition the assumptions regarding simplicity of class definitions hold. The remainder of the cost is amortized over subsequent data accesses (see appendix D). This includes two major categories of operations:
- Merging of CMInts which are no longer differentiated on the basis of descriptors, or conversely, splitting of newly differentiated CM-Ints, and
- the redirection of the links between Class Containers and TIRs through the union-find scheme to enable access to classes in a single step from TIRs and vice-versa.

When an existing attribute dimension is split, or an entirely new dimension added to the AR-Tree, in the course of reformulating class definitions, existing CMInts must be reclassified into entirely new TIRs. This implies complexity proportional to the number of objects in the newly constituted class. Efficient structures for such cases need further research.

### 7.6.1 Choices in Interval Tree Structure

The efficiency of searching the AVInt and CM-Int Trees for a particular segment (time interval) is determined by:
- cost of locating the node where the desired segment is located, multiplied by
- the cost of searching the 2d tree indexing the segments stored at the node.

These costs are affected by the choice of time points labeling the AVInt Tree nodes, and may be motivated by two possibly conflicting criteria:
- to label the nodes of the tree regularly in time. This makes the complexity of locating the node negligible, but can cause asymmetry in the distribution of intervals among the nodes. In turn, the search of the secondary structure can be $O(\log(n))$ in the number of intervals (if all intervals are at one node).
to distribute the intervals uniformly among the nodes of the tree. This reduces the
search cost of the secondary structure (O(1) if the number of intervals per node is
bounded by C), but needs an O(log(h)) search time if h is the number of nodes.

These goals are interdependent, since the second option reduces to O(log(n)) if h = n/C,
where C is the bound on the number of intervals at a node. In the average case where the
interval lengths and distribution are uniform, search takes sublinear time in the number of
intervals. As a compromise, we propose that the number of intervals per node be bounded
by C, and that the time points labeling the nodes of the tree be neither regular nor random.
Instead, they should split the time space in powers of 0.5 as required. This bounds the search
of Endpt.Trees and yet allows them to be identifiable easily. The AVInt Tree and the CMInt
Tree are assumed to be balanced off-line, making that cost irrelevant to our bounds.

7.7 Chapter Summary

Chapter 7 proposes some original hybrid data structures derived from work in computa-
tional geometry for the conceptual organization of data in dynamic and adaptive informa-
tion systems. Computational geometers have proposed efficient data structures for interval-
and polygon-related operations such as retrieval of all intervals (partially) covered by a
given interval. These techniques are useful in the context of our proposal where interval-
based attribute value timestamps are used, and class membership is deduced. We propose a
unique hybrid of these data structures, make them dynamic to enable changes in class defi-
nitions, and apply it in an original way to dynamic and adaptive information systems.

Our contributions yield the ability to support the usual operations alongside class-
based and interval-based queries. They also support queries on previous and future class
memberships and are capable of being efficiently reorganized to support changing class defi-
nitions. Through these capabilities, they support the claim that queries and updates in
dynamic and adaptive information systems can have acceptable efficiency. It is important to
clarify that these data structures are limited to supporting changes in schema consisting of
collections defined by hyper rectangles in an n-dimensional attribute space. They also do not
claim to be optimal for each of the operations that they support. Hence, they provide a step-
ning stone to further research.
Chapter 8  Adaptive Information Systems: Extending Object Dynamics

In the previous chapters, we have proposed modeling primitives and a computational framework for reasoning with object dynamics in some detail. This chapter, whose contributions are secondary to those of chapters 3 to 7, proposes an extension of the previous results pointing towards interesting areas for future research. Our contribution is a set of modeling devices and terminology for reasoning with schema under the ATN framework, and illustration through example. Since alternative schemas of a domain are possible, we propose a set of primitives with minimal semantic commitments, which isolates the constraints imposed by data models and users. The constraints applicable in reasoning with schema are then categorized, we show by example how they are expressed, and how changes in preference or data can violate them to trigger a schema change. We also exemplify constraint interpretation by ATNs in order to enable reasoning with changing schema. Constraints on schema being more complex than those on objects, we propose higher order representation, and also suggest that reasoning with schema should be driven directly by the utility of the current schema.

A body of knowledge $K$ (including pre-suppositions) should correlate with a body of questions $Q(K)$ that can be posed on its basis. A question $Q$ belongs to $Q(K)$ if $K$ assures that $Q$ can have an answer by satisfying all of $Q$'s presuppositions regarding existing objects and constraints on them. As argued in chapter 1 and appendix A, the means of reasoning with presuppositions and whether or not certain schemas satisfy those of a set of questions should lie outside of the language of the schema itself. Accordingly, we propose that adaptive information systems should have extra-schematic means to establish and change presuppositions. With respect to a certain schema, certain questions can not be answered (or certain updates can not be effected) because the schema does not adhere to the presuppositions of the question or update. In addition to the option of rejecting the query or update, adaptive information systems allow an administrator the option to specify how the schema may evolve in order to adhere to the presuppositions.

The classifications and methods encapsulated by a schema $K$ are determined by the need for questions and updates to be $K$-relatively meaningful [133]. Questions and updates based
on a schema are determined by the questioner's values, which provide for the rationalization of her actions by promising a beneficial state of affairs. The values of the user of an information system are approximated in our proposal by the utility of the schema with respect to the terms of the question or update. Utility scales can be cardinal or ordinal, monopolar (0 to infinity only) or bipolar (- to +) closed-ended versus open-ended etc. In the case of objects evolving over time, utility was defined as a binary entity dependent upon whether or not the definition of the class was satisfied. In the context of changing schemas, we propose that the optimal schema with respect to a set of questions or updates be that which satisfies the largest consistent set of pre-suppositions (entities and constraints) of the expected questions or updates, and enables the satisfaction of other non-functional constraints like efficiency. To simplify the problem, we assume the goodness of schema in adaptive information systems is evaluated against a single set of criteria instead of many sets from a community of users.

8.1 Alternative and Changing Views on Data

Two sets of questions can legitimately pre-suppose different viewpoints on a domain, neither of which is inherently better without reference to the use of the information. For example, viewpoints from research, engineering, and inventory on the same information may discern entirely different sets of entities from the same raw information.

A critical requirement in emerging applications is the ability to make dynamic changes to the representation of enterprise knowledge in response to external events such as market conditions. Creative, design-type applications, which demand interfaces to information which adapt to changed priorities, are now larger users of databases. We may extrapolate to visualize increasingly personalized schemas derived from a common pool of enterprise data, dictated by the needs (such as size, level of detail and execution time) of each user in terms of her knowledge and the nature of her queries. As the knowledge level and focus of the use change, so will the nature of queries, the terms of reference, and consequently, the schema.

Such a scenario is enabled by the concept of adaptive information systems introduced in this chapter. The existing research on integration of heterogenous database schemas targets specific models independent of user-requirements. On the other hand, there exists independent research on user-models and abstraction. Adaptive information systems extend the
grammar-based formalism for dynamic databases for integrating and evolving of heterogeneous schemas based on user requirements. Pictorially, the scenario is shown in figure 8.1, which depicts the movement of data among databases, and the reconstitution of schemas based on conceptual considerations. As the user's preferences and needs change over time, a common customizable data model should be interpretable (as opposed to translated) to the terms of the desired models as depicted in Figure 8.2.

Figures 8.1 and 8.2 illustrate the changes in views over time and the generation of views at a specific point in time, respectively. Deciding the transformability of two unconstrained schemas or data models is unsolvable in general, because finding a mapping (unifying) function for two sets of first order schema constraints from the domain of all possible functions is a true second order problem.

Even if constrained, finding a constructive solution to the problem of evolving a schema in response to changes in user requirements can be intractable. In the simplest case,
the problem reduces to relational view creation and reconciliation. The adaptive information systems problem is differentiated by the following:

- The set of models is not homogenous, as in the relational view management.
- Schema changes and data migration are driven by changes in the users' preferences regarding appropriate abstractions, and data distribution over time.

The ontology is less restricted than in the view management problem: base entities, attributes, and classes are not predetermined.

8.1.1 Comparison of Object Dynamics and Schema Adaptation

Dynamic object migration and adaptive schema evolution are strongly interdependent. Except in trivial cases such as the acquisition of a factual attribute by a class, schema evolution always triggers changes in the classification of existing objects. If a class disappears as a recognizable entity, its instances either become classified in the observer’s view in a different class, or generate objects in other alternative classes. Similarly, object evolution may lead to a (user-mediated) schema evolution. When an object acquires some attributes which cause it to be a bonafide member of more than one class, the user may be forced to re-organize the class hierarchy. Other researchers have proposed another kind of interrelationship in the form of relativity of schema and instance, where one and the same object can act as an instance of a class, and at other times, act as a class which has instances (e.g. [50], [159]).

Preference-driven schema adaptation can be formulated as a goal-driven planning task. The schema is incrementally evolved in response to changing preferences, as in the case of a planning task where state is modified in response to a combination of desired goal-state and/or incremental change in context. The problem of reconciling an existing schema with a 'goal' schema dictated by a set of user requirements may be seen as an E-unification problem with the equational theory specified by a grammar. A grammar-based description of schema change is better than atomic change specifications found in planning frameworks, because:

- A single (ATN) computational framework may be used to reason with schema or instance, depending on how a particular entity is interpreted by the observer.
- Restricting the grammar makes the complexity of schema changes controllable.
- ATNs are expressive enough to cover a range of schema transforms similar to the range of object evolutions that can be defined.
Deductions involving past states of schema and different viewpoints is possible, over both lossy and lossless transformations.

Triggering of interdependent schema and object changes is made easy.

Users' requirements can be captured in the model through preconditions and postconditions, allowing pro-active and customized schema transformations.

Preconditions -> based on current and historical attribute values of instance 1, and global conditions

Actions -> implement the updates necessary to implement the reaction

Postconditions -> integrity constraints to be maintained by instance 1 and instance 2 after the reaction, and/or by other objects. Derived from class types and adjectives.

Preconditions -> based on current and historical set of attributes of class 1 or relation 1, and user preferences

Actions -> implement the changes necessary to enable the schema change

Postconditions -> The rules to be applied on the constraints already existing among the meta-attributes to find new constraints to be obeyed by class 1, class 2, relation 1 and relation 2

FIGURE 8.3 Comparing Object and Schema Dynamics

For example, the constraints placed on the evolution of schema may imply one or more of the following:

- Losslessness in transforming structural object-oriented schema to relational ones.
- Preservation of value-based keys, i.e. if there is a set of attributes which uniquely identify objects of a certain type, this property would prevent the schema from changing so that members of resulting object type(s) are not distinguishable by their attribute values.
- Preservation of extraneous (non-intensional, or incidental) keys in any evolution.

Figure 8.3 summarizes the similarities and differences among object migration and schema change. Metaclasses, or classes of schema elements, are useful as abstraction mechanisms for classes, for the same reasons as classes are useful for instances. The design of schema change can integrate static and dynamic interactions in the same way as in the case
of classes. Similarly when a class is reclassified from one abstract metaclass to another, it
carries the analogous class attributes with it in much the same way as an instance carries its
instance attributes.

8.1.1.1 Similarities Between Object and Schema Dynamics
In object migration, when the attributes of an object change, it might migrate to become
another type of object, or cause the creation of another object. In schema change, when the
set of services or interfaces required from an information management system change, it
might change into another schema, or might create another (perhaps co-existing) schema.
Links might be maintained among source and target schemas for reasoning. For example:

- Classes of objects are distinguished by definitional attributes. Similarly, classes of
  schemas are distinguished by the kinds of meta-attributes a schema has, and the
  kinds of constraints that apply between them. Non-1NF relations, for instance,
  are less constrained than a 1st Normal Form relation.

- Classes of objects can acquire/lose/modify factual attributes without affecting
  the class they are in. Classes of schema elements can similarly acquire / lose / modify
  their set of incidental attributes without affecting their essential properties. If, in R(a,b), b is functionally dependent on a, and no other constraints apply, then there is no fundamental change if this is changed to R(a,b,c) where c is functionally dependent on a, and no other constraints apply.

- Most of the pre-conditions, post-conditions and actions can be automatically
  generated given the transition and attribute types, in both cases.

- Reversibility is a relevant for schema changes as well as instance migrations. At
  the schema level, every schema transformation which is done to unify two data-
  base schemas must be reversible, and all the updates must be propagatable to the
  original schemas.

- M:N changes are relevant at both levels. At the schema level, in the simplest case,
  one relation may be split into 2, with certain integrity conditions maintained
  between them.

- Spontaneous transitions must be supported at both levels, since a change in the
  user's requirements should spontaneously lead to the addition or deletion of cer-
  tain information from the schema relevant to the user.

8.1.1.2 Differences among Object and Schema Dynamics
Crucial differences with object migration include the fact that schemas change less fre-
quently, and when they do change the mapping of the constraints in one schema to another
is a more difficult problem. Many aspects of schema change are administrator-mediated and must be decided at runtime, in contrast to the case in object migration. For example:

- Instances are typically classified into a rich set of classes, each with a reasonably simple structure, and relatively few reactions. Schemas typically involve a sparse but more complex set of classes with a rich schema transition structure. In contrast to the simpler structure of schema histories, the structure and constraints associated with schemas themselves are usually more complex than those for instances. Whereas the instances of a class (e.g., "Person") are relatively uniform in structure, the instances of an abstraction mechanism (e.g., "class") are diverse.

- Preservation of identity is a relevant issue at the instance level. At the schema level, almost all changes are generations (perhaps accompanied by the survival of the source schema, as in the case of instances).

- Reversibility in schema dynamics carries the connotation that integrity of the source and target schemas can be maintained through updates to both source and target schemas after the reaction, apart from the ability to restore the old schema.

- An update to an instance of a metaclass which does not cause it to evolve into a different kind of abstraction mechanism is nevertheless significant. In contrast, an update to an object which does not cause it to change classes is less significant.

- Temporal reasoning is likely to be more important for instances than for schema; more instance queries are likely to access the temporal dimension.

- The structure of object histories is often more complex than expressible by regular languages, as illustrated earlier. In contrast, it is difficult to justify a more complex structure for schema histories than that offered by regular languages. In general, if changing conditions dictate a change in schema, no commitment to future changes is made, unlike in the case of instance histories.

- Whereas object migration usually involves definite reasoning, adaptive schema change involves utility-based reasoning as in planning. Hence, there is usually a set of weighted choices to be made at every stage of schema evolution.

**8.1.1.3 Reconciling Similarities and Differences**

The differences lead us to propose distinct sets of entities to define the dynamics of instances and schema. However, the underlying similarities suggest that the computational mechanisms underlying both instance and schema reasoning should be based on a uniform framework, in our case the ATN.
8.1.2 Framework for Adaptive Schema

In object dynamics, we assumed that the schema, classes and transitions were fixed. Schema change makes these elements dynamic; classes and transitions can be added and deleted from the schema. To aid in this effort, we start with the following commitments:

- Every schema element (class, transition, relations between classes, attributes) of a database schema is an instance of a metaclass (an abstraction mechanism).
- Changes in the user's preferences or changes to other schemas may trigger changes pro-actively. They may also be done manually by an administrator.
- These evolutions lead reversibly or irreversibly to the creation of new schema classes and transitions with new constraints. The system enables reversibility.
- When a schema change occurs, all instance of the antecedent schema need not eagerly undergo an object transition. This may be done lazily, on demand.
- At any given time, the collection of all instances of the meta-classes, along with the applicable constraints, constitute the current database schema. Likewise, the collection of desired instances of meta-classes, along with their constraints, constitute the desired database.
- Type of metaclasses, unlike instances of metaclasses, and instances of classes, cannot be created, deleted or modified dynamically (within the scope of the framework in this thesis).

8.1.3 Criteria for Schema Adaptation

In d'ialect(o), classes are defined intensionally in terms of a set of constraints on attribute values. An object belongs to a class if it obeys the constraints. A reaction may happen if some of those constraints are violated (and others become obeyed). The same mechanism founds adaptive information systems. A data model provides certain abstraction mechanisms, which are used as templates to model an application domain. Such abstraction mechanisms are characterized by their adequacy and expressive ease [124] for a particular task. We define the concept of meta-properties that describe these qualities of the instances of abstraction mechanisms. Constraints of different kinds apply to these meta-properties.

Analogously, change in these meta-properties of a schema could induce an instance of an abstraction mechanism to adapt by undergoing a schema change reaction yielding one or more instances of (perhaps distinct) abstraction mechanisms that better satisfy the constraints imposed by data models or applications.
A simple example of this scenario is when information must be presented at different levels of abstraction for planning at different depths of detail. For example, in a communications management system at a coarse level of abstraction, planning goals and constraints can be set for the entire collection of service circuits in an exchange. At a finer level of detail, those goals can be translated to individual circuits and specialized parameters. In this context we may consider that the schema may adapt to the desired level of abstraction through the application of the abstract function below:

- \( \text{Make Schema}(\text{Queries: Domains} \rightarrow \text{Objects}) = \text{Desired Schema} \) : a function which derives query presuppositions in terms of the sets of objects that are distinguishable, and what constitutes an entity for the user. It uses a measure of the difficulty of evaluating such queries on different types of objects to yield an acceptable schema.

For instance, a valid input is "queries are for input impedance. The entities for which input impedances are desired should be distinguishable by the circuit board types (individual boards need not be distinguished)". The schema for this user need not have extraneous information like board-identification (i.e. is at a coarser level of abstraction).

The resulting schema may then be integrated with schemas for other users.

8.2 \text{D'Dialect(S) Assumptions and Primitives}

\( \text{D'dialect(s)} \) is a framework for defining, and effecting changes in, static conceptualizations of domains. It provides a set of concepts in terms of which abstractions of domains may be specified. In order that the abstractions used to model a domain (the objects, classes, attributes, relations etc.) are flexible enough to adapt to their users’ preferences and domain conditions, the founding commitments about the building blocks of a domain model should be more finely granular than objects and groupings, and less semantically committed.

Our basic standpoint is that any static snapshot of a world which is axiomatized through a first-order theory implicitly reflects some perceptual bias, in the choice of entities, functions and relations in the domain of discourse (the ontology - we refer to [29] for definitions of first order languages, interpretations, theories and their models). Our interest lies in making explicit and capturing those biases that make an axiomatization \text{useful} in \text{d'dialect(s)}.

\text{Example 8.1} (Diversity in representation) If an observer feels the need for the existence of:
· persons
· vehicles
· an ownership correspondence between persons and vehicles
· constraints on ownership - a person owns at most one vehicle, joint ownership is allowed, and all vehicles must be owned.

(Note: (i) We avoid saying “one-to-many function from vehicles to persons”, because from a data modeling perspective, that is one of many descriptions.

(ii) The existence of persons and vehicles as classes is not a given. They exist if it is useful from the users' perspective for them to exist in the axiomatization.)

she may model the domain using one of the following alternatives, among others:

1. a class Persons, a class Vehicles, and a partial functional attribute
   Owns: Person --> Vehicle

2. a class Vehicles, a class Persons, a class Owner-Group of disjoint subsets of Persons, and a single-valued function
   Owners: Persons --> Owner-Group.

3. a single 'relation' (null attribute values allowed)
   Vehicle-Ownership-Records-of-Persons.

The specific choice of modeling strategy adopted is determined by, among other things, the capabilities of the representational tools, the requirements in terms of the kinds of facts which we desire to infer from the model, and the amount of effort which can be invested in the process of inference. From another perspective, commonly accepted interpretations of syntactic notations need not necessarily apply, as in the following example.

Example 8.2 The assertion Person1 (Name1: John, Age1: 19), commonly has the following presuppositions: there is a type of things called Person1, one of whose instances is a thing whose attribute Name has the value 'John' and whose attribute Age has the value '19'.

Alternatively, Person1 could be viewed as an instance of the relationship Person between the string 'John' and the integer 19. A particular relationship instance is valid if 'John' is related to Name1, an instance of the concept Name and 19 is related to Age1, an instance of the concept Age.

As the examples above illustrate, alternative representations may be equally acceptable depending on extra-domain properties. In order not to pre-determine distinctions among the
building blocks of representations, we base adaptive representations on the single uniform constructs of unit and association, which have minimum semantic commitments [70].

**Definition 8.1** A *unit* is a symbolically representable information entity. A unit is distinguishable from other units and has a unique identity.

The definition of unit does not imply atomicity; units need not be the most basic pieces of information. Units (similar to intensions of [50]) are capable of being combined with others to form larger information aggregates. The basic form of combination is the association.

**Definition 8.2** An *association* is an unordered n-tuple of units with some perceivable connection or relationship in the domain. Associations have identities and act like units in other associations.

Associations have been explored in [70], and later in [101] as a basis for hiding the ontological distinctions among atoms of information under a heterogenous universal relation formalism. Associations are distinguished from one another by the restrictions placed on them.

**Definition 8.3** A *restriction* is a general constraint expression capturing the nature of the relationship among the units in an association.

Restrictions are similar to undirected constraints [85] in a higher order logical context. Defining associations as n-tuples with restrictions enables set or class relationships to be derived from observation, rather than embedded by axiom, as follows.

The observable building blocks of the domain are captured as units, without predetermining what units are relatively atomic with respect to others. The observable relationships among units, without prejudicing the semantic primacy of any one kind of relationship over another, are captured by associations. The primacy of experience, and the perception of relationships, are determined by the purposes and abilities of the individual perceiver. *The units and associations are transformed into instances of specific abstraction mechanisms by the process of imposing commitment* (through User-Dictated restrictions- section 8.3.1) onto the basic model.

### 8.3 Abstraction Mechanisms

The basic building block of adaptive schema is a first class entity called *abstraction mechanism*. An abstraction mechanism provides templates in terms of which the types of any
domain of discourse may be described. Abstraction mechanisms provide mechanisms for
different types of restrictions to be made explicit and be maintained, thereby enforcing disci-
pline on the description and evolution of domain descriptions. Depending on the kinds of
relationships which need to be represented regarding a domain for a particular observer,
particular abstraction mechanisms may or may not be adequate.

**Definition 8.4** An Abstraction Mechanism is a collection of named, typed
Attribute Roles. An instance of an abstraction mechanism in a
schema is called an Abstraction. An abstraction can Manifest
other abstractions in an attribute role associated with the
abstraction mechanism of which it is an instance. A manifested
abstraction is a Member of the attribute role in which it is
manifested.

Two abstraction mechanisms are distinguishable if their respective sets of attribute
roles are not identical in name and type.

**Definition 8.5** An Attribute Role imposes two types of constraints on its
members:

- **Range Restriction:** This constrains the type of abstraction mechanism
  which can act as a member of the attribute role.
- **Role Constraints:** This constrains the kinds of relationships which a
  member of the attribute-role can have with the members of other
  attribute roles.

**Example 8.3** "1st normal form relation" is an abstraction mechanism offered
by the relational model. "Object" and "class" are abstraction
mechanisms offered by object-oriented models. An abstraction,
'Person', may be modeled as a 1st normal form relation if it has
no complex attributes and has a key. It may be modeled as a
class if each instance has an identity, complex structure and
needs to be encapsulated. An attribute role of 1st normal form
relation is Key. The Person abstraction could manifest the
abstraction Name (which is an instance of the abstraction
mechanism Attribute) for the attribute role Key. Name is a
member of Key for the abstraction Person. A range restriction
constraint on Key might be:

Each member must be a singleton instance of the Attribute. This
enforces the absence of multipart keys.

Examples of role constraint on Key may be:

- Uniqueness of values for members of Name for all instances of Person
- Functional dependency of every member of attribute role Properties on Name

We assume that distinct attribute roles have unique names, i.e. if a particular attribute role (say "key") is present in two abstraction mechanisms, it is assumed that the range restrictions and role constraints associated with 'key' in the two cases are identical.

An abstraction can be a member of more than one attribute role of another abstraction, and an abstraction can be manifested in attribute roles of more than one other abstraction subject to the manifested abstraction obeying all applicable constraints. I.e. Name can be a member of Key as well as Properties, and it appears in attribute roles of Person as well as Vehicles.

Notation: The Applicable Constraints on an abstraction A with respect to another abstraction B is the set of range restrictions and role constraints defined on A as a result of the fact that B manifests A in some (possibly more than one) attribute role(s).

Definition 8.6 The Schema Dynamic Properties of an attribute role of an abstraction mechanism is a set of properties which determine the part it plays in affecting schema change. They comprise:

- its Type
- Dynamic Structural Constraints: Constraints which are independent of the individual members of the attribute role, but determine the semantics of all members of the attribute role under change

Note that the type associated with each attribute role of an abstraction mechanism may differ in different abstraction mechanisms in which it occurs, unlike the attribute role's range and role constraints. We consider types and dynamic structural constraints in section 8.5.1.

Example 8.4 Key is generally definitional in type with respect to the Relation abstraction mechanism, but incidental with respect to the Class abstraction mechanism. If the members of Key are changed, it is more significant in a relation than it is in a class. A Dynamic Structural Constraint may specify that a relation can have at most 3 keys, each being a single abstraction. If this structure is violated, a schema change may be triggered.

Definition 8.7 An attribute role \( a \) is dominant (directly or indirectly) in an abstraction \( A \) with respect to another abstraction \( B \) if the members of \( a \) in \( A \) constrain the members of another attribute role \( b \) in \( B \) through a sufficient but not necessary role constraint.
Definitions of subordinate and equal (or co-dominant) attribute roles follow naturally from the above definitions. Based on these definitions relating attribute roles to abstractions, the concepts of domination/subordination/equality among abstractions emerges as follows: if an attribute A has an attribute role a which is dominant with respect to an abstraction B, and B has no attribute roles which are dominant with respect to A, then A is dominant with respect to B. Subordination and equality (or co-dominance) among abstractions emerge analogously. We consolidate the concepts regarding structure of schema elements and higher order constraints, by substantiating the intuitions of previous examples in example.

8.3.1 Restrictions in relation to Schema Adaptation

From the viewpoint of a user, an adaptive schema should evolve in accord with allowable rules of change between known abstraction mechanisms when there is a mismatch in the relationships and restrictions captured by abstraction mechanisms of the current schema, and those which are implied in the domain of discourse and the user’s view of the domain. In the terms of section 8.2, when a restriction associated with some association of units is violated, a different set of associations may be required among the different units of information representing a domain. This requires reasoning with different types of restrictions. We categorize the restrictions in dialect(s) as follows:

- **Model-Inherent restrictions**: These define the data model in terms of the abstraction mechanisms offered and their semantics. For example, they specify the structural and use aspects of IS-A links and non-1NF relations. Model-inherent restrictions define abstraction mechanisms by associating range and other restrictions on attribute roles and templates of the role constraints. Addition of a new kind of attribute role means a change in the model itself. We defer the issue of reasoning with the addition of entirely new attribute roles.

- **Domain-Inherent restrictions**: These represent the set of associations dictated by the domain. Domain-inherent restrictions instantiate range and role constraint templates of the abstraction mechanisms. They must be consistent with the model-inherent restrictions of the abstraction mechanism and obey the dynamic structural constraints. Common examples include observed functional dependencies between units, disjointness of domains etc.

- **User-Dictated restrictions**: These reflect the observer’s perspective on the domain. The observer’s knowledge and goals determine:
  - the basis on which objects can be distinguished from one another,
- the basis on which they may be grouped into query domains (classes),
- the terms of the queries which may be posed on these domains, and
- the desired characteristics of the answers such as size and the aggregation functions to be used for particular attributes (max, min etc.).

These restrictions determine usability, and therefore evolution of schema. Within the bounds of the domain inherent and model-inherent restrictions, they instantiate role constraints and dynamic structural constraints of the abstractions.

Pictorially, the process of converting one schema into another in terms of these restrictions is shown by example in figure 8.4.

![Diagram](image)

**FIGURE 8.4** Abstraction Mechs and Meta-Reactions in terms of Restrictions

Role constraints which capture the model-inherent restrictions are of a higher order with respect to the domain-inherent and user-dictated restrictions, since they are quantified over all predicates and functions referred to in the domain- and user-inherent restrictions. In order for the formulae in this language to be decidable, however, the domain of the higher order variables must be restricted. Thus, it is only for notational convenience that we adopt a higher order logic with first order semantics (e.g. HiLog [44] or F-Logic [94]) as do [92] to express these constraints. This is reasonable, given that the higher order variables may be instantiated by constants from a restricted domain of predicate and function constants.

**Example 8.5** An abstraction mechanism Limited_INF_Reln manifests the following attribute roles:

- *Key*
- *Properties*
- *Instances*
· Size (used in example 8.6)

For simplicity, we give only the role constraints on the attribute role Key.

1. The range restrictions on Key is that its member must be an abstraction of type Single-valued attribute.

2. There are five role constraints:

2a. There must be exactly one member of Key:

\[
\text{constraint : Limited}_1\text{NF}_1\text{Reln}(R) \Rightarrow (\text{Key}(R) = f(R))
\]

\[
\text{constraint : } [\text{Limited}_1\text{NF}_1\text{Reln}(R1) \land (\text{Key}(R1) = K_1) \land (\text{Key}(R1) = K_2) ] \Rightarrow (K_1 = K_2)
\]

2b. The member of Key is also a member of Properties:

\[
\text{constraint : } [\text{Limited}_1\text{NF}_1\text{Reln}(R) \land (\text{Key}(R) = K) ] \Rightarrow (K \in \text{Properties}(R))
\]

2c. Instances manifest the properties of the Limited_1NF_Reln:

\[
\text{constraint : } [\text{Limited}_1\text{NF}_1\text{Reln}(R) \land (K \in \text{Properties}(R)) \land (I \in \text{Instances}(R))] \Rightarrow (K(I) = X)
\]

2d. Values of Key must be unique among the Instances of Key:

\[
\text{constraint : } [\text{Limited}_1\text{NF}_1\text{Reln}(R) \land (\text{Key}(R) = K) \land (I_1 \in \text{Instances}(R)) \land (I_2 \in \text{Instances}(R))] \Rightarrow [ (K(I_1) = K(I_2)) \Rightarrow (I_1 = I_2) ]
\]

2e. All members of Properties are functionally dependent upon the member of Key:

\[
\text{constraint : } [\text{Limited}_1\text{NF}_1\text{Reln}(R) \land (\text{Key}(R) = K) \land (I_1 \in \text{Instances}(R)) \land (I_2 \in \text{Instances}(R)) \land (P \in \text{Properties}(R)) ] \Rightarrow [ (K(I_1) = X_1) \land (K(I_2) = X_2) \land (P(I_1) = Y_1) \land (P(I_2) = Y_2) ] \Rightarrow [ (X_1 = X_2) \Rightarrow (Y_1 = Y_2) ]
\]

All of the above are model-inherent restrictions placed by the abstraction Limited_1NF_Reln above on its instances. There may be a similar abstraction mechanism Class with fewer restrictions on Key, but with additional attribute roles such as Superclass.

Note that the terminology f(R) is used in (2a) to refer to a Skolem function of R, or an existentially quantified variable which denotes a key attribute. Note also that the terminology of the form X(Y) = Z is used above as a shorthand for "there exists an association between units X, Y, and Z, which is interpreted directionally as Y has a property X whose value is Z".

The domain inherent restrictions on a particular world can include associations among units of information, such as the following:

1. n1: (Name, "John"), a1: (age, 33), s1: (SIN, 22), a2: (age, 22), n2: (Name, "Buick"), t1: (10, time), x1: (20, x_coord), y1: (30,
y_coord), h1: (brown, hair_colour) ... which signify "type declarations" with minimum commitment, like 'the information unit "John" and the information unit name are associated in the domain', 'unit 20 and unit x-coordinate (in a cartesian frame of reference) are associated in the domain' and so on. Note that each association is labelled with an identity so that it can be used in other associations.

2. o1: (t1, x1, y1, n1, a1, s1, h1), o2: (t2, x2, y2, n2, a2, s2) ... are associations of information units in the domain. These express the domain inherent restriction that the associated units are "observable together" in the domain. (This denotes the collection of experience that is common in stating either one of "rabbithood again" or "rabbit frame in time" - see chapter 1.)

3. Observed constraints : An arbitrary domain-inherent restriction on pairs of associations (X, age, Y) and (integer, Y) may be "Y < 100". This is an arbitrary constraint that may be true in a particular domain. More ordinary observations include those of the following type :

(t, x, y, n, a, s, h) ∧ n:(Name: "John") => h:(brown, hair_colour) ∧ s:(SIN, 22)

or "Whatever else happens, the (name, John), (brown, hair_colour), and (SIN, 22)" are observed together. In the most familiar interpretation : "John is one of those lucky people whose hair colour and SIN never change in time or place". Similarly, statements such as "everyone always ages with time" and "birthdate is today's date minus age"... can be domain-inherent restrictions.

The domain does not determine an observer's view of it autonomously. Whether or not the unit brown is recognized as a value of the attribute hair_colour or the attribute 'brownness' whose value includes hair_colour as in "the 'brownnesses' of John include his hair_colour and eye_colour" can not be an observer-independent decision. Indeed, the status of "John" as the entity to which properties are attached is not an observer-independent decision. Enumeration and correlation of observations is, however, observer-independent. What the observer must add to this is her preferences for observing this information and querying it from a pragmatic perspective, as shown in example 8.6.

Example 8.6 From a pragmatic perspective, the observer may have the following (alternative) sets of preferences/knowledge :

1. "I recognize all spatially co-located properties to be manifested by a single object"
2. "I recognize all properties associated with the same string in
association with the unit ‘Name’ to be a single object”

- These would depend upon whether or not the concepts of location and/or name are recognizable or relevant to the observer. The preferences are expressed logically as:

1. \( \forall x \forall y \exists z (\text{Entity}(z) \land \text{Association}(x \_ \text{coord}, x) \land \text{Association}(y \_ \text{coord}, y)) \), or simply \( f(x, y) \leq (x \_ \text{coord}, x) \land (y \_ \text{coord}, y) \). This statement would objectify everything associated with a particular spatial co-ordinate, even though separated in time or other ways.

2. \( \forall x \exists ! z (\text{Entity}(z) \land \text{Association}(\text{Name}, x)) \), or simply \( f(x) \leq (\text{Name}, x) \). This statement would objectify everything associated with a particular name, even though separated in time or spatial co-ordinates.

- Taking the second alternative as our thread through this example, the next set of preferences concerns what attributes the observer would find of interest in the selected entities. These may be derived as follows from the expected queries which may be expressed as “SELECT <attributes>”, and extracted according to the formula:

\[
(a \in <\text{attributes}>) \land \text{Association}(z, t) \land (t:(a, x)) \Rightarrow (a(z) = x).
\]

Hence:

a. \( \text{Name}(z) = \text{N} ; \text{age}(z) = \text{A} ; \text{SIN}(z) = \text{s OR} \)

b. \( \text{brown}(z) = \text{B OR} \)

c. \( \text{hair\_colour}(z) \)

The attributes to be evaluated against the objects may require the specification of functions derived from domain-inherent restrictions or specified. For example, birthdate may be calculated in days from age and current date, since birthdate is not a directly observable quantity. Similarly, if the chosen object were “board type” and an attribute were “board area”, the board area for that type may be found by averaging the area of the instances (section 8.1.3).

Following option (a) above, the Make_Schema function derives the following types of structured entities from the domain inherent associations. Notice that entities such as “Buick” might fall out of the group on account of not having SIN:

\( (f(\text{"John"}), (\text{Name} \rightarrow \text{"John"}), (\text{age} \rightarrow 33), (\text{SIN} \rightarrow 22)) \)

- The last revealed observer-preference in structuring the representation concerns the grouping of the structured entities into query domains. This may be driven by the user’s requirements regarding the size of the grouping (derived from the query performance requirements) and/or the frequently needed selection criteria as per the “WHERE <condition>” clauses of
queries. For instance, if:

i. age < 30 is a frequently cited condition because certain queries are evaluated only on youth, and

ii. SIN < 50 is a good criteria for splitting the group of derived entities into two equal sized groups, given that no group should exceed 3 members. The resulting groups of entities may be:

I. (f("John"), (Name -> "John"), (age -> 33), (SIN -> 22))
   (f("Jim"), (Name -> "Jim"), (age -> 45), (SIN -> 25))

II. (f("Jill"), (Name -> "Jill"), (age -> 23), (SIN -> 55))
   (f("Jane"), (Name -> "Jane"), (age -> 13), (SIN -> 72))

III. (f("Janet"), (Name -> "Janet"), (age -> 21), (SIN -> 21))
    (f("Janet"), (Name -> "Janet"), (age -> 23), (SIN -> 11))

Thereafter, these groups of derived entities must be modeled in terms of certain abstraction mechanisms, as allowed by the schema change description. We resume this series of examples in section 8.5, in the context of meta-reaction types.

Different units of any domain of discourse stand in specific relationships to one another (as specified by domain inherent restrictions). A subset of these units and inter-unit constraints is relevant to any particular user and/or acceptable to the abstraction mechanisms offered by any particular data model). A particular abstraction mechanism is appropriate to model the domain of discourse if its members offer a set of attribute roles such that all and only the relevant constraints from the user and domain’s point of view are captured in mapping domain elements to particular instances of the abstraction mechanisms.

8.3.2 Concealed Schema

In adaptive information systems in contrast to dynamic information systems, we may assume that certain constraints may be maintained by hidden parts of the schema which are not visible to users. For example, the concept of time may not be explicit in some model, but implicit in an attribute role with role constraints which impose a total order on the members of a hidden height abstraction. This is illustrated in example.

Example 8.7 In case the following entities and their attributes are required by the observer:

(f(Name), (next_taller -> X))
but if the domain-inherent constraints support the statement of facts of the kind:

\((f(\text{Name}), (\text{height} \rightarrow \text{Y})) \text{ only,}\)

the question of supporting the user's perspective based on a fact that must not be visible to the user arises.

Two perspectives may be taken:

1. Precompute the next_taller relation based on height and remove the height attribute. In addition to losing some information, the next_taller attribute must be a member of a distinguished attribute role, with distinct update semantics.

2. Retain the height relation in a hidden attribute role, and evaluate the next_taller relationship based on height at runtime. An update to the larger abstraction will require a calculation of a pseudo-value for the height attribute which satisfies the constraints of next_taller.

Thus a schema change reaction may require commitment on the part of the system, to maintain some schema elements which are not directly accessible to an observer. The mechanics of denying access to the observer, and of calculating pseudo-values, are a part of the implementation domain for each data model and are omitted from this thesis.

8.4 Abstraction Dynamics and Meta-Reactions

Given the categorization of the constraints and the way they are used in the context of changes to schema, we now introduce the modeling devices which are instrumental in capturing allowable changes to existing schema.

Instances of abstraction mechanisms can be transformed to instances of other abstraction mechanisms through meta-reactions. The constraints implied by the restrictions named above are translated into pre- and post-conditions on the meta-reactions, in a way similar to objects. When the pre-conditions are violated and others are satisfied, schema changes may be effected. A schema constituent is an instance of an abstraction mechanism if it obeys the constraints implied by the model-inherent, domain-inherent and user-dictated restrictions. Changes to these restrictions may be provoked by changes in the requirements of applications using the data in terms of the domains over which queries will be formulated, the functions to be evaluated, and the factors which distinguish objects. These restrictions may also be violated by changes in the data distribution in the domain.
Schema change frameworks can be defined in terms of abstraction mechanisms and meta-reactions linking them. They allow administrators to define the acceptable schema types and the kinds of schema changes allowed within an enterprise database composed of a set of heterogeneous databases. In the interest of schema relativization, some researchers have proposed operators which are identical for both schema and instance levels [159]. However, these proposals account only for the structural aspects of schemas and instances. When both structural and behavioural constraints are considered, it seems artificial to define a uniform set of primitives for the twin purposes of object migration and schema adaptation. We believe that certain meta-reactions which are relevant to schema are not relevant to instances and vice-versa. The same is true for attribute types and classes. Accordingly, we define distinct sets of primitives, united through the common computational framework of ATNs.

8.4.1 Dynamic aspects of Abstractions

The schema dynamic properties which determine how a schema behaves under change depend upon the types of the attribute roles and the dynamic structural constraints associated with them. A preliminary taxonomy of types in d’dialect(s) follows (this is a topic of further research). The classifications are motivated by the preceding discussion on dominance and user-dictated restrictions.

Types distinguish attribute roles from 3 perspectives:

- The effect of a change in observer’s preferences on the member abstractions.
- Effect of a change in an abstraction in that attribute role on the rest of the schema.
- Effect of a change in the schema or database instance on meta-attributes.

Figure 8.5 presents our current view regarding classification of attribute role types.

- Categorization based on dominance determines whether a change to the attribute role affects other abstractions and if so, which types of abstractions are affected
- Visibility of an attribute role, explained in example 8.7, determines cencalment of schema. “Height” may be a member of an attribute role which is inaccessible.
- Factual attribute roles do not affect schema changes. For example, the addition or deletion on the Properties attribute role in example 8.5 does not affect the Limited_1NF_Reln abstraction mechanism or its members, since properties are factual. The Key attribute role, however, is definitional. Definitional attribute roles may lead to schema change in two ways:
• If the range or role restrictions of a Truth-Factor is violated, the definition of the abstraction mechanism is violated by its instance, triggering a meta-reaction that may cause the instance to become a member of another abstraction mechanism. This is the case with the Key attribute role.

• If a similar constraint of a Use-Factor is violated, a more global evaluation of alternative schema options results. An example of a use-factor is relation-size (example 8.6) which prompts a re-evaluation of the grouping of entities and the discovery of an attribute on the basis of which a grouping may be split.

• Extensional attribute roles are those whose members' values are directly available from the domain-inherent restrictions. Intensional attribute roles are those whose members' attribute values are derived from other, perhaps hidden attribute roles, as is the case with birthdate and next-taller in the foregoing examples.

![Attribute Role Types](image)

**FIGURE 8.5** Attribute Role Types

Other than type, the **Dynamic Structural Constraints** associated with an attribute role decide how an abstraction might change with user perspective and data. Role constraints are associated with an attribute role no matter which abstraction mechanism it is associated with, while dynamic structural constraints are peculiar to an attribute role in a particular abstraction mechanism.

Dynamic structural constraints are likely to be simpler than role constraints. A set of specific constraint types as below may suffice for describing dynamic behavior of schema.

• *Fixed-Members*: This dynamic structural constraint implies that if the members of the attribute role change, a meta-reaction may be triggered. It is applicable in data warehouses, where entity-sets keyed on attributes such as medical insurance number or driver's licence number may be distinguished from each other.

• *Fixed-Member-Value*: This type imposes an additional constraint over Fixed-Member attribute roles, stating that a specific member abstraction is required in that
attribute role. It is useful in situations where universal identifiers based on a specific attribute such as SIN are useful. Absence of SIN might trigger a change.

- **Presence:** Presence constraints are exemplified by cardinality constraints on constituents of `Central_OO_Schema`, `Non_1NF_Schema` etc. shown in section 8.5.1.

- **Functionally-Defined:** Constraints relating attribute roles using set operators (e.g. `Key <= Properties`, example 8.5) or mathematical constraints on the cardinalities of the attribute roles.

Though still a topic of research, we hypothesize that these dynamic structural constraint types are sufficient to express the constraints on most schema change descriptions.

### 8.5 Schema Change Reactions

Named meta-reactions connect abstraction mechanisms as in figure, and specify the allowable options for changing the schema in reaction to changes in the observer's requirements and domain-inherent restrictions. As in object dynamics, meta-reactions are classified into a small set of well-defined types usable by an administrator as a tool for specifying valid schema transformations. As in object migration primitives, the incompleteness of these or any other limited set of primitives with respect to all possible schema changes is inevitable. Since we propose to interpret these primitives computationally in terms of ATNs, the extensibility of these primitives within known trade-offs of computational costs and expressive power is assured, as in the case of our primitives in d'dialect(o).

The question in schema evolution is: "Given some user-dictated and domain-inherent conditions which must be met by the schema, is there a derivation path from the current abstraction to one which satisfies those conditions?". The path is composed of a sequence of elementary meta-transitions.

**Definition 8.8** A (named) **Schema Change (meta-transition)** is a dynamic relationship between two abstraction mechanisms, the **Source Abstraction** and the **Target Abstraction**, at a specific time. A schema change optionally has pre-conditions, actions, and postconditions (constraints to be maintained in the future).

Terminology similar to groupware research is used to categorize meta-transitions.

**Definition 8.9** A **Schema Evolution (Version Update)** is a schema change where the source abstraction and target abstraction have the
same identity. The attribute roles, types and dynamic constraints associated with the source abstraction cease to be valid after the meta-transition, while those associated with the target abstraction are valid at and after the change.

The semantics of a version update specify that any other abstraction which depended on the source abstraction refer through to the target abstraction after the version update event. When reference is made to instances of the changed class at a time prior to the meta-transition, the source abstraction constraints remain valid. However, the attribute roles and dynamic constraints associated with the source schema are not maintained for those reactions that occur after the schema change. A version update can trigger schema changes in related abstractions.

**Definition 8.10 A Schema Enhancement (Concurrent Version Creation)** is a schema change where the source and target abstractions share the same identity. The attribute roles, types and dynamic constraints associated with both source and target schemas continue to be maintained after the schema enhancement.

The semantics of schema enhancement imply that source and target abstractions exist concurrently. However, the roles associated with the source (resp. target) abstraction are not significant when the target (resp. source) abstraction is used. In practical terms, schema enhancement implies the maintenance of two representations of an abstraction (perhaps in two different data stores under different data models). The framework ensures that changes made to an instance of the source (resp. target) abstraction do not affect that instance in its capacity as an instance of the target (resp. source) abstraction.

**Definition 8.11 A Version Supercession (Version Checkpoint)** is a schema change where the source abstraction ceases to exist. Instead, a new abstraction (version) starts to exist with a different set of roles and constraints at the time of the meta-transition.

Practically, this models version deletion in groupware applications. All references to instances of the deleted version dangle after the point of the reaction. Version checkpoint may prompt other schema changes or object migrations. The instances of the source abstraction undergo transformations to the target abstraction. The abstractions manifested in the attribute roles of the source abstraction may or may not continue to exist depending on their other attribute role memberships and the nature of the binding (deep/shallow).
**CHAPTER 8. ADAPTIVE INFORMATION SYSTEMS**

**Definition 8.12** A *Version Derivation (Downward Commit)* is a schema change where the source abstraction and target abstraction do not share identity, and the source abstraction persists after the version derivation. The target abstraction is created as a stand-alone abstraction.

This meta-reaction is useful in groupware design applications where a designer may withdraw a design from the common source and create a private sub-version. The new version is not linked to any other abstraction within the framework (the user application may explicitly link it to another abstraction).

**Definition 8.13** A *Version Reconciliation (Upward Commit)* is a change peculiar to schemas which involves merging two schemas which had earlier participated in a downward commit, while preserving specific constraints on the separate version paths.

We do not discuss in detail the computational interpretation of these semantics in terms of the ATN, other than by example. Suffice it to say that given an expression of the constraints in a higher order logic (with first order semantics) based on register values, and the ability within the ATN formalism to attach conditions to registers, the ATN framework is capable of the task.

![Hypothetical schema change description](image)

**FIGURE 8.6** Hypothetical schema change description

With the aid of these meta-transitions to relate abstraction mechanisms, we resume our thread of examples from section 8.3. The emergent entities, their properties and their groupings had been derived from the domain-inherent and user-dictated restrictions in examples 8.5 and 8.6. The remainder of the schema change function consists of finding which abstraction mechanisms and meta-transitions are ideal for representing and changing the informa-
If the schema change description is as given in figure 8.6, the groupings arrived at could be modeled either as a set of Limited_1NF_Relns, or as a set of Objects. Having described the model inherent restrictions of the former in example 8.5, we outline the reasoning by which the derived sets (I, II, and III of example 8.6) may be interpreted as Limited_1NF_Relns.

- Sets I and II are similar and are discussed together. Since the meta-reaction is a version checkpoint, there are no considerations regarding preservation of constraints applicable prior to the change. We assume that the actions associated with the meta-reaction ensure that all the properties are instantiated into the attribute role Properties associated with Limited_1NF_Reln. (This is true if “Previous Schema” also had an attribute role called Properties, which we assume for simplicity). Similarly, the attribute role Instances is initialized with the appropriate sets of objects as applicable. Candidates for the Attribute role Key include all members of Properties viz. \{Name, age, SIN\} as per (2b). (2a) states that exactly one of these must initialize the attribute role Key. (2d) and (2e) are satisfied by the Properties and Instances of both sets I and II. (2c) is defined appropriately. Given the fact that any of the attributes can serve as the key, the observer may be prompted or an automatic choice made at this point regarding the Key attribute role.

- Set III is identical except for the fact that Name can not satisfy the constraints (2d) and (2e) because the domain-inherent restrictions specify there to be two “Janet”s. At this point, the administrator or observer may be prompted for an alternative Key, either for the set III alone, or for all the three sets.

Therefore, the meta-transition results in the creation of 3 Limited_1NF_Relns, and a valid Limited_1NF_Schema. If the alternative meta-reaction is taken, to yield an object-oriented schema, the Key constraints would not be important, but the superclass/subclass relationships inherent in the OO model would have necessitated the creation of a structure as in figure 8.7, where the dotted shapes denote abstract classes, the solid areas denote concrete
classes, and the directed arrows denote subclass relationships. The relationships between the partitions of the "Youths" set for evaluating queries are assumed to be set up appropriately.

8.5.1 Overall Schema-Structural Change

We view a database schema as an abstraction mechanism with a complex structure, defined similar to the previous section. For example, a relational database is an abstraction mechanism which has an attribute role set of relations, which is a set of relation abstraction mechanisms. Each relation has attribute roles such as the key used in examples above.

In grammatical terms, we view the evolution of entire schemas to be defined by an attribute grammar (similar to [156]) where the role constraints are encapsulated in the inheritance and synthesis functions associated with symbols of the grammar. We deviate from Tan and Katayama's approach in the way the evolution of the schema over time is modeled (it is not modeled in their system). Allowable changes to the schema may be specified in terms of the d'dialect(s) primitives which are interpreted by a grammatical description, analogous to object migration. The concept of migrating a distributed heterogenous schema as a whole is illustrated in figure using a simplified example of allowable dynamic schema behavior.

![Figure 8.8](image)

Figure 8.8: Data model evolution driven by incremental schema change

The figure depicts a hypothetical schema change description where Limited_1NF_Schemas are defined in terms of Limited_1NF_Relns. Limited_1NF_Relns may react in M:N fashions so as to produce more or less numbers, but the schema type remains unchanged. Limited_1NF_Relns may react to make NFNF_Relations. A collection of NFNF_Relations defines a Non_1NF_Schema (note that a mixed schema is undefined). A Central ОО_Schema is defined in this framework as a collection of Limited_1NF_Relns,
NFNF_Relns and OO_Classes, so long as there is at least one of the latter. A Central_OO_Schema may change into a Distrib_OO_Schema if it joins an aggregate of more than one Central_OO_Schema and adds a Distribution_Function. Notice the number of loops and density of the connectivity that we anticipate from our limited study of the requirements of systems which would specify schema change pattern.

8.5.2 Computational Aspects of Adaptive Schema

The computational framework (ATN) underlying both object migration and schema change are similar in motivation and structure. They differ in the complexity and expressiveness of the constraints, and therefore, the difficulty of the reasoning involved.

A schema is defined as an aggregation of the set of attributes attached to the nodes of the attribute grammar, extending our formulation for objects where object structures are defined by their attributes attached to the nodes of the attribute grammar. Role constraints are captured by the inheritance and synthesis functions attached to grammar symbols. The dynamic behavior of the schema over time is captured in the trees and sentences produced by the ATN interpretation of meta-reactions, in analogy with reactions in object migration.

In computational terms, if the user’s requirements or domain-inherent restrictions change, some pre-conditions on an ATN link are violated, leading to a derivation which implies a change in the schema. However, the computational issues for schema are different:

- The constraints on schema are more expressive than for instances. The reasoning involves higher order constructs such as $f(R)$ (where $f$ is a function variable in -2a-, example 8.5) and $K(I_1)$ (where $K$ is a predicate variable on members of the attribute role Key in -2d-, example 8.5). However, the domain of unification for these variables is finite, making the logic first order, with a higher order syntax. Moreover, the user is involved in sanctioning schema change where second order concepts are involved as in the case where $f$ is a function variable in “next_taller(X) = f(height(X))”.

- Reversibility is costly, since every reaction that has occurred since a schema change must be reconsidered if it is reversed. Links between source and target schema may be maintained, but they are not so useful for system-controlled reversibility. Instead, reversibility of schema change means a physical restoration of the previous schema.

- Whereas efficiency is important in object migration, properties such as ambiguity are more relevant to schema change (to ensure that there are no two ways to arrive at the same schema, and conversely, to ensure that two schemas do not satisfy a set of requirements equally well).
The preconditions that determine whether or not a meta-reaction is triggered are based on the (loss of) utility of the source abstraction. (Reactions are formulated alike, but utility is bipolarly equated to adherence to the source class definition - section 4.3.1). Utility drives schema change and unlike object migration, it is not bipolar (section 8.6).

### 8.6 Schema Change Reaction Determinants

Changes in schema are not governed by non-adherence to the definition of an abstraction mechanism. Rather, it is governed by the utility or usefulness of a schema with respect to a certain set of operations (query and update) and knowledge of the user. For instance, if a database update adds 2 elements to Set \( S \), violating the Use-Factor Size for the Youth-part2 abstraction (which is an instance of the Limited-INF-Reln abstraction mechanism, it does not immediately invalidate the abstraction, but makes it less desirable by violating a non-functional constraint. When an observer's perspective or domain-inherent restrictions change in a way that requires the re-formulation of the representation, the question being asked is:

"What is an optimal schema (considering costs of query evaluation, schema transformation etc.) where the following conditions are satisfied? : function \( A_i \) on entities distinguished on the basis of \( B_i \) and \( C_i \) and grouped on the basis of \( E_i \) and \( F_i \). (\( X_i \%) of the queries)"

Syntax for expressing such requests is outside the scope of this thesis, as is the derivation of the specific conditions from the model of a user's goals, knowledge etc. However, it is evident from the request (optimal schema, ... costs ...) that the decisions with dynamic schema are based on cost, rather than concrete meta-class definitions.

In the remainder of this chapter, we illustrate the application of value-based measures in an ATN-based framework for adaptive information systems, and discuss the properties that such measures should satisfy in order to support utility-based schema change.

### 8.6.1 Utility applied to Adaptive Schema

We refer the user to appendix B for a short discussion of utility theory. The measure of a schema's suitability for an application depends upon, among others:

1. Efficiency of evaluation of the functions \( E_i \) relevant to the application. This may be a function of the number of joins required on average (relational), or the number of pointers to be chased (network). Efficiency can be influenced by storing intermediate
views, or actualizing parameters by evaluating the constraints at the time of schema change, rather than runtime.

2. How close to the user's intuitions the objects in the database ontology are ($N_a$). For example, if the user would like to query population of municipal divisions, but the schema recognizes political constituencies, there is a measure of mismatch.

3. How similar the user's and the database's categorizations of the world are ($S_b$), in terms of the recognized subdomains as per criteria such as generalization, aggregation and abstraction.

Therefore, utility is a factor of the following numbers:

- A per-data-operation cost differential ($C_o = f(E_f, N_a, S_b)$) derived from the measures of discrepancy between each existing schema element and the abstractions reachable by effecting a meta-transition.

- A one-time measure of difficulty ($D$) in transforming the schema according to a specific meta-transition, which may be or may not be (observer's discretion) amortizable over all the operations that would be performed on the resulting schema ($D$).

We must rely on a utility differential function of the form $u = g(D, C_o)$ to compare which of the available meta-reactions is warranted in the context of a given change in observers' or domain characteristics. Hence, a change of schema has a basis in maximizing the utility of the information with respect to its users' needs. Additionally, in the case of schema change, utility must play a role in making a recognizer deterministic and unambiguous, in ensuring monotonic increase of utility through a series of changes. Under certain assumptions, a user's preferences between alternative schema can be characterized in terms of some properties of these alternatives. The utility measures relevant in object migration and schema change should have the following characteristics:

- The utility of each abstraction is an aggregate function of the utility of its sub-parts.

- The subparts may not be independent of each other with respect to the utility measure. For example, a certain join needed to find an attribute efficiently may need all of a set of intermediate relations to exist, not a subset.

- In the case of schema, the utility depends on the expected value of the schema to answer a particular mix of question types. The strength of preference for a schema may be a function of the relative importance of each question in the set, as well as the relative difficulty of answering each question.

- The set of choices in alternative representations is finite or countably infinite, never nondenumerable.
We discuss the question at the base of the use of utility to direct schema change: the nature of the utility function \( f \) and preference relationships in alternative schemas \( E_f, N_a \) and \( S_s \) reachable through different meta-reactions. All of the assumptions above are justifiable if \( C_o = f(E_f) \), ignoring the other variables, and in this case, the existence of a function \( U \) is assured (see appendix B). In the presence of the other variables, it is clearly unreasonable to expect preference among schemas to be a strict order. Schema preference is difficult to cast as a weak order because 'negative transitivity' is not assured in the context of subjective preferences for certain abstractions over others by the observer. Another choice is to make preference a strict partial order. This view allows indifference to be non-transitive, since \((x=y, y=z, x<z)\) is an allowed state of affairs. Under this scheme, indifference is not an equivalence relation, but a different equivalence relation may be defined (identically indifferent) as follows:

\[
x \sim y \iff (x = z \iff y = z, \forall z \in X)
\]

\( x \) and \( y \) are identically indifferent if they are indifferent with respect to the same set of choices in \( X \). Though plausible as a characteristic preference relation, it is difficult to gauge the usefulness of this form of indifference for preference among schema for the purposes of deriving a utility function. In between these two, there are interval orders which attribute an intermediate degree of strength to the indifference relationship, which we ignore.

In summary, we assume the possibility of a weak order being imposed on \( N_a \) and \( S_s \) which would allow a weak order to be imposed on preference of alternative schemas, so that a utility function would exist between non-indifferent choices. This leaves the question of the function \( f \) for aggregating overall utility from that of its constituent parts. Additive utility measures are unrealistic in the case of schema change, because in most cases the utility of a schema depends upon the evaluation efficiency of certain mixes of queries and updates. Evaluation efficiency is often a multiplicative or exponential function of the sizes of the information involved and there are multiple ways in which each such query can be answered. Hence, even if \( N_a \) and \( S_s \) can be additive, \( E_f \) can be multiplicative in terms of the same components that are used in calculating the former measures.

To summarize our impressions of the utility measures and functions that may be used in aiding decisions regarding schema changes in adaptive information systems:
• The utility of a schema with respect to a set of queries depends on the entities and groupings in the candidate schema, and the efficiency of operations on them.

• The preferences of abstractions and groupings may be subjective to the observer. It seems reasonable to suppose that a weak order can be imposed.

• The expected utility of a schema in terms of the efficiency of operations on its elements is difficult to calculate because:
  • of the difficulty in deciding the independence of queries from one another. At present, we assume that independence exists.
  • of the non-linear nature of the cost functions for evaluation of particular operations.

• If the above are resolved, then a weak order on preference may be imposed on alternative schemas, and in this case a utility function is guaranteed to exist.

• In case the schema change description is simple enough to be expressible as a finite automaton, the measure above would lead in a straightforward way to a criterion for defining collection classes resulting in a deterministic version of the finite automaton.

Specific utility functions for adaptive information systems is a topic of future research.

8.6.2 Schema change using ATNs and Utility

We illustrate the use of utility, higher order logical reasoning and ATN-based grammars in reasoning with schema change through the example in figure 8.9.

FIGURE 8.9 Schema Change through violation of a Use Factor

The figure depicts the ATN links modeling meta-reactions that would allow an instance of a limited 1NF relation to become either an OO class or split into N limited 1NF relations by becoming a "Split_1NF_Reln" abstraction mechanism, which is aggregated of N "Limited_1NF_Reln"s. (We use a shorthand notation to depict the aggregation. The figure
also shows an existing abstraction, and its instances, and the addition of a new fact (domain-
inherent restriction) in an instance of an existing Limited_1NF_Reln. The higher order def-
nition of the "Limited_1NF_Reln" is as follows:

$$\forall R \text{reln}(R) \leq \exists K \forall P \forall X1 \forall X2 \forall Y1 \forall Y2 \text{(R(K(X1),P(Y1)) \land R(K(X2), P(Y2)) \land (Y1=Y2)) \Rightarrow (X1 = X2)}$$

The formula above defines limited 1NF relations as pairs of unary predicates on units, one called K, the other called P, such that if the arguments of P are identical, then the argu-
ments of K are identical as well. R K and P are higher order predicate variables, but first order semantics is applicable because the range of all three is limited to the set of units, and not the set of any arbitrary function or predicate on all units. Translated to HiLog in terms of Prolog as per the rules in [44] and standard texts on Prolog (see Clocksin and Mellish, "Programming in Prolog", appendix B), the definition results in the following clauses (symbols that are capitalized denote variables, others denote constants):

$$\text{reln}(R) :\text{all}(P, \text{all}(X1, \text{all}(X2, \text{all}(Y1, \text{all}(Y2, \text{keyclause}(R, K, P, X1, X2, Y1, Y2)))))$$

$$\text{keyclause}(R, K, P, X1, X2, Y1, Y2) :\neg \text{R}(K(X1, P(Y1)); \neg \text{R}(K(X2), P(Y2)); (Y1 <> Y2); (X1 = X2).$$

$$\text{X(Y)} : \text{association}(X, Y), \text{query_attr}(X).$$

$$\text{R}(X1(Y1), X2(Y2)) :\text{X1}(Y1), X2(Y2), \text{association}(X1(Y1), A), \text{association}(X2(Y2), A), (X1 <> X2), \text{reln_name}(R).$$

We omit axioms such as commutativity for brevity. The predicates in bold font denote preferences that are derivable from a user model regarding which attributes will be queried, and which unit may represent the grouping of elements in the domain. These may be derived as shown in earlier examples. The solvability of such formulae by an engine (such as Prolog III) that an interpret the symbols ";", "all", "not" and higher order symbols is assured.

Links are traversed based on utility. The utility of every instance of the abstraction mechanism "Limited_1NF_Reln" may be predicated upon use_factors (e.g. size) and truth factors (related to the definition). In terms of our discussion, Co = \(f(Ef, Na, Ss)\) defines utility before the cost of schema change is considered. Since a Limited_1NF_Reln has no subparts, it is plausible that \(C_o = \text{Defn} \ast (Ef \ast Na \ast Ss)\), i.e. the utility is a multiplicative factor of the effi-
ciency of function evaluation, and the suitability of the abstractions and structure. Defn is a truth factor which is 1 if the definition of the abstraction mechanism is obeyed, and 0 other-
wise. Again, since 1NF relations have no structure, Ss may be fixed as a constant 1. Na may
be simplistically formulated as the ratio of abstractions that match the user's `reln_name` and `query_attr` conditions. In our example, assuming that "person" and "age" are the only desired abstractions in their respective categories, \( N_a = 1 \). The use-factor \( E_f \) may be defined as \( 1/\text{size} \), so that it decreases as the size increases.

Under the scenario shown, "person" continues to obey the definition of the Limited-1NF_Reln abstraction mechanism making \( \text{Defn} = 1 \), but its size increases, leading to a decrease in the utility, and the possibility of an ATN link being traversed. Analogously, the utility of "person" as an instance abstraction of the OO_Class or Split_1NF_Reln abstraction mechanisms may increase. The one whose utility is greater would determine which of the meta-reactions is effected after accounting for the cost of effecting the change. The above exemplifies the merging of ATNs with utility and higher order logic for schema change.

### 8.7 Chapter Summary

Chapter 8 proposes terminology and concepts to extend our proposed object migration framework for the purpose of effecting changes to schema in response to changing needs and information. Its contributions do not extend beyond the proposal of these concepts and their illustration by example. We regard this chapter as a stepping stone to future research in the area of schema change based on the proposed framework.

The dynamic behavior of schema in response to changes in the needs of the observers is different from that of the dynamic behavior of objects as a result of changes in domain attributes. We therefore propose a distinct set of dynamic primitives to model the behavior of schema, along with a basic terminology for defining schema so that dynamic reasoning may be performed with it. The primitives separate the constraints implied by specific data models and user preferences from the basic observable units of information and associations among them. This separation allows us to reason with the constraints as user preferences and data distributions change. Constraints on schema are higher order with respect to the constraints on objects. However, we propose that they can be interpreted through a first order semantic model in order to be tractable.

Since we believe that usefulness of the representation strongly influences whether or not it should change, utility is integrated into the framework. Underlying this new set of primitives, we claim that our ATM-based formalism with supporting data structures is sufficient to execute queries and updates on schema, as in the case of objects.
Chapter 9  Conclusions and Future Work

We started with the viewpoint that information often exists in order to perform diagnoses based on past information that has changed, and to predict into future states. We justified the need to capture the dynamic relationships underlying reasoning with changing information and schema declaratively, and to provide a computational framework for dynamic reasoning as a part of queries and updates. Our objective was to propose a comprehensive framework for information systems that represent the dynamic aspects of objects, and allow schema to adapt to users' needs over time. Within the context of such a framework, this thesis has outlined user-interface languages to model object migration and schema change with good expressive coverage of the domain of evolving systems. Though the user-level modeling primitives for object migration and schema change proposed in this thesis are distinct, we have proposed a single computational model for implementation and analysis, because of the underlying similarities.

Real-life object migration and schema evolution requirements are diverse. This leads us to propose an extensible computational framework, rather than attempt to define a single large set of primitives and try to justify its completeness. The computational framework is designed to be extensible with "adjectives" that capture additional semantics, with accompanying trade-offs between expressibility and performance. The computational framework is based upon well-understood results from formal language theory. Hence, properties of dynamic descriptions, such as the complexity of evolution and ambiguity of object histories, fall out as meta-properties of the underlying formalism. Other inferences implied by the semantics of the query language are realized within our framework through a set algorithms for parsing and generating strings of object states in the language defined by the grammar. Though the algorithms are derived from established results, their adaptation to the context of dynamic domains is unique to our proposal.

This is a systems-architecture thesis, motivated by modeling problems in the use of databases for representing and reasoning with domains which involve change. Despite the emphasis on modeling, implementability is crucial. In order to address this issue, we have adapted work in computational geometry to propose a set of hybrid data structures which
allow acceptable performance by the reasoning algorithms. We do not claim optimality for these data structures, and acknowledge that the data structures should be extended to store objects with complex structures. However, they contribute a starting point in the area of data structures capable of efficient reasoning with changing objects and schema definitions, specially the latter. Briefly, they address object migration and schema change for classes defined by iso-oriented hyper-rectangles in the attribute space.

In each of the areas addressed by this thesis, there is potential for further study, since we have generated many more questions at every step than answers.

9.1 Future Work

Some questions thrown up by our study of this practically motivated area include:

- What are the strong links between static and dynamic modeling? We have initiated discussion on the nature of integrated static-dynamic modeling in chapter 5, and have suggested that they can not be formally deterministic, since modeling involves some "art". However, we feel that more concrete axioms may exist, at least in the context of particular domain types.

- How can data structures geared towards adaptive schema be optimized?
  - Real world classes are rarely definable by iso-oriented hyper-rectangles. For example, a class \( x \) may be defined by the condition: \( x.A + x.B < P \), where \( A \) and \( B \) are attributes. Such conditions define non-iso-oriented polygons which may require structures such as cutting trees. Controlling the complexity of these structures is an immediate area of research.

- Attribute values can vary continuously through time, as in the case of temperature in a manufacturing process. Our assumption of piecewise discrete attribute values and histories thus needs to be generalized. Histories and attributes defined by continuous lines in time and attribute space ultimately need to be modeled.

- How should our formalisms be extended for schema change? Some definitions have been presented, and a few mechanisms extended in chapter 8, but this area is of active interest to our future research. The extension of the algorithms, query language and data structure for designing flexible schemas is planned.

- What are the implementation parameters for dynamic and adaptive systems? For example, the set of segments indexed by each range tree is arbitrarily restricted to an individual objects' attributes. It may be more useful to index the attributes of groups of interdependent objects through a single tree. Design choices such as the usability of segment trees also need further research with more data.
• How might the class definitions and attribute value changes introduced in this thesis be extended within the bounds of acceptable complexity? The allowable class definitions and patterns of change in attribute values given in this thesis are weak. Extensions may include class definitions and constraints involving negation, disjunction or existentials. Attribute values often change as a function of time, a possibility which we have ignored.

• What meta-properties of formal languages are relevant in describing dynamic systems? For example, different grammar structures (e.g. CNF/GNF/String grammar etc.) seem to suggest intuitive distinctions among real world systems, though these forms are expressively equivalent.

• Are finer distinctions among formal properties of grammars beneficial to dynamic modeling? For example, grammatical ambiguity seems to need finer classification, perhaps in terms of a metric similar to graph connectivity. Such a metric may help to define the degrees of structural freedom in structures such as plans, in the context of their grammatical definitions.

• In our experience, most dynamic definitions involve context-sensitive features. What are the relevant meta-properties of context-sensitive systems in general? Similarly, what meta-properties in the context of derivation trees may be used to describe dynamic features such as deadlock? Deadlock is a complex topic in itself, because the determinants involve complex dependencies among the internal attributes of states.

• How might the issue of historical archival be addressed by our data structures? The arities of the trees influence the ability to balancing them under insertions and deletions, as well as the ability to archive subtrees efficiently. This issue is of practical relevance for the usability of historical systems, since the volume of data is large.

• What other interesting domains in the real world can these techniques be applied to? We feel the need for validation of our proposed architecture against a broader range of candidate domains.

While results pertaining to some of the topics above will be addresses in the course of future research on our part, we feel that practical compulsions will drive the need for dynamic and adaptive databases faster than the results can materialize. In the near future, as more processes are made re-engineerable for competitive advantage in industry, and more heterogenous database systems are required to work together in enterprise knowledge bases, the issues addressed by this thesis are likely to gain importance. We hope that our investigations can provide useful leads into the realization of systems addressing these needs as the opportunities arise.
Appendix A: Change and Perspective in Philosophy

A.1 Causality in Philosophy

Causality is exemplified by laws such as: “if A occurs in a replicable state B, then state C always results”. There is strong debate about the standing of such a notion of causality, and causal reasoning, in the domain of discovery of natural laws. Two relevant debates are:

- A-priori status of causality: Rationalists have argued that causality must exist as an a priori assumption. Others such as Kant, “... agreed with (Hume) that causal connections were not so much discovered by our minds in the world, as imposed by them on the world, (but) regarded the principle of causality as a regulative principle of reason, to be justified by argument ...” [117]. Under the rationalist scheme, analytic descriptions of ‘causes’, and a logic based on the ‘causes’ connective are indicated. This view is more relevant in scientific explanation involving natural kinds. In information systems, the kinds and their states are defined by the needs of the application, making strong causal relationships hard to defend.

- Replicability, and locality of effect: The assumption that causal relationships among objects and events are influenced only by their limited spatio-temporal vicinity is debatable in general. The central question is: “Is any inductive hypothesis possible about a change in state of an object without considering the global state of the domain?”. Though interesting in the philosophical domain, we recall that information systems model idealizations of the domain of discourse. In this context, we can safely assume that granularities over which similarity is judged are chosen so as to approximate replicability ([117], pp 277-293).

Informational entities being nominal (not natural) kinds, causal relationships between states of existence are difficult to justify, but legal relationships among states can be hypothesized. What aspects of causality may apply to legal relationships? In the empiricist view,

1. causality is not given by the senses, as for example, color is.
2. causality is not particular in the sense that it is not dependent on identities of objects.
3. causality is not deductive; it does not follow from the definitions of natural kinds.

Of these, 2 and 3 are of particular relevance to us. The positive definition states:

1. Causality is a necessary relation between cause and effect in the modal sense $1\neg A \Rightarrow 1\neg \neg QA$. Coincidence should not be confused with cause.
2. Causality is weakly antecedent. Cause could be coincident with the effect, but not subsequent to it. The cause is the least set of sufficient conditions needed to achieve the effect.

3. It is a repeatable relationship.

A.2 Empirical Observation and Belief

Causal relationships governing change are relative to an observer’s perception of a domain, her “common sense” about the domain. However, in [117] (page 5) we read: “A well-recognized feature of common sense is that ... it seldom is aware of the limits within which its beliefs are valid or its practices successful. ... common-sense knowledge is most adequate in situations where a certain number of factors remain practically unchanged ... ”. Since information abstractions are “organized common sense”, the same observation applies:

- The factors on which the domain view is predicated should be recognized externally as they change, in order that the abstractions remain compatible with the goals of the user.

Empirical observation and ontology have seen lengthy debate in philosophy. Traditional thinking (Schwartz, [144]) is founded on the intensions of terms. Members of the terms’ extensions are determined by their adherence to the intensional definitions, which could be a strict conjunction of terms (Russell, Frege) or a cluster of part-definitions (Wittgenstein, Searle). The new thinking (e.g. Putnam and Kripke) on this subject is distinguished from the traditional in that proper names are seen as primary descriptors in their own right. Names act as rigid designators across all possible worlds. The notion of rigid designators is carried over to common nouns and natural kinds like ‘tiger’ and ‘gold’, based on the fundamental properties of the referenced objects, such as the atomic number of gold or the genetic composition of tiger. Though useful for natural kinds with some extra-logical basis in biological, physical or chemical phenomena, this thinking does not seem to fit as well the nominal kinds (Locke), things in the world which are defined in terms of the natural kinds. Information abstractions in enterprise databases are largely concerned with nominal kinds.

A.2.1 Foundations of Empirical Observation and Belief

In determining representations, if each postulated entity or observation must rely for its justification on a prior observation, then we are faced with the impossibility of finding a suitable bottoming out of the beliefs. The traditional approach has been foundationalism,
which forces this regression of justifications to a halt by postulating some beliefs to be basic and not needing justification. The arguments against this position are based on the arbitrariness of these unquestionable foundations. In information modeling, the foundationalist position is restrictive, particularly since most of the types are nominal (though information systems' domain is much shallower than reality).

The main counterpoint to the foundational school is provided by the coherentist school. Coherentists postulate that the infinite regress of empirical justification ends in a "closed curve of some sort and that justification ... is rationally cogent, rather than vitiated by circularity". "... inferential justification ... is essentially systematic or holistic in character: beliefs are justified by being inferentially related to other beliefs in the overall context of a coherent system." ([28], page 90). The underlying observations must have the following character:

- A system of beliefs is coherent only if it is logically consistent.
- A set of beliefs is coherent in proportion to its degree of probabilistic consistency.
- The coherence of a system of beliefs is proportional to the number and strength of the inferential connections between them.
- The coherence of a system of beliefs is decreased in proportion to the presence of unexplained anomalies in the believed content of the system.

The most contentious point for convinced coherentists (Rescher, Lehrer, Blanshard among them) concerns the question of associating truth with coherence.

### A.2.2 Ontological Relativity

Coherence (through compensatory adjustments within a set of terms) is adopted as a basis for empirical observation by Quine in [168] through the notion of Ontological Relativity. Quine suggests that a kind names a set of objects which are internally cohesive or coherent to a greater degree than to any set belonging to another kind because of some situation-dependent criteria. The notion of representational flexibility is extended to great lengths in the essay "Ontological Relativity". For example, Quine suggests that the reaction to the objective presence of a rabbit could well be "a rabbit", or "rabbithood again", or "5 consecutive rabbit-frames-in-time" or even more outlandish categorizations of the world such as the verbification of rabbit, "it is rabbiting again" ("Speaking of Objects", in [168]). In a similar way, all of water, red and rabbit could be perceived as the same 'kind' of things, provided compensatory adjustments were made. This is illustrated through a linguistic example, viz.
the translation of “ne ... rien” in French to either of “not ... anything” or “nothing” in English, Quine extrapolates this concept to the topic of describing rabbits in different terms.

Quine discusses the fallacy involved in imposing “the one true” standards of objectification on other perceivers: “The implicit maxim guiding (a field linguist’s) choice of ‘rabbit’... is that an enduring and relatively homogenous object, moving as a whole against a contrasting background, is a likely reference for a short expression. ... one of the linguistic universals. ... But the maxim is his own imposition, towards settling what is objectively indeterminable.”. Quine’s claim is even more plausible for the limited case of nominal kinds than for the natural kinds. Two observers can perceive the same situation (in the view of an external observer) in terms of two differing sets of terms if the situations are removed from physical or biological ‘undisputables’, and closer to the observers’ own.

A.3 Pragmatic Coherence

In the context of these arguments on the variability of ontologies, we visit briefly the problem of choosing between alternative coherent system. BonJour [28] reviews some of the main approaches to ‘truth’ in coherent systems as follows. Neurath and Hempel would choose the “most coherent” in their positivist theory, allowing a well-written novel to define the choice. The idealist position of Blanshard would rely on the most globally coherent system, an approach that would cause practical difficulties in terms of evaluability. Lehrer’s subjectivist coherence would choose a set of basic beliefs which are filtered to eliminate all those which might bring the observer any personal gain.

Rescher’s pragmatic coherence account takes the opposite view to Lehrer’s position. Rescher’s criterion is the measure of practical success achievable by the adoption of a coherent account of the world. Rescher claims that “inquiry procedures which systematically underwrite success-conducive theses thus deserve to be credited with a significant measure of rational warrant.. This agrees with the purpose of information modeling: the achievement of practical ends - it takes into account the purpose for which the truth is accumulated, or the reasons for its documentation and existence.

Though the enterprise of information modeling does not pursue truth, Rescher’s criterion is the one which inspires our own to the largest extent, as we address the issue of choosing one from among a set of possible representations of a domain, as dictated by the user’s needs.
Appendix B: Utility

B.1 Basic Utility Theory

Preference is a relation between entities, which may be reflexive, transitive, symmetric, antisymmetric according to the usual definitions. A relation is negatively transitive if \( \neg xRy \land yRz \Rightarrow \neg xRz \). A relation is connected or complete if \( \forall x,y \in Y (xRy \lor yRx) \). A relation is weakly connected if \( \forall x,y \in Y (x \neq y \Rightarrow (xRy \lor yRx)) \). In terms of these definitions, the properties desirable of preferences between two alternative abstraction mechanisms seem to be as follows:

- Preference of all kinds should be irreflexive, antisymmetric and transitive. A schema or object state can not be preferred over itself, two entities can not be preferred over each other, and if one is preferred over another, which is preferred over a third, then the first should also be preferred over the third.

- Indifference towards alternatives should define an equivalence relationship for schema. The practical implication of this statement should be that if \( \{x\} \) is an equivalence class of schema under indifference, and \( x \) and \( y \) are members of \( \{x\} \), then if there is a schema change from \( x \) to a schema \( z \), there should also be a change from \( y \) to \( z \). This is in analogy with the collection classes defined to convert to deterministic finite automata.

Definition B.1

1. A weak order is a relationship \( R \) on \( Y \) which is asymmetric and negatively transitive.

2. A strict order is a weakly connected weak order.

3. An equivalence relationship is reflexive, symmetric and transitive.

4. A strict partial order is a relation which is irreflexive and transitive.

For example, \( '<' \) is a strict order (and a weak order) on real numbers. \( '=' \) is an equivalence relationship. If \( '<' \) denotes the relationship of strict preference and \( '=' \) denotes the relationship of indifference (i.e. \( \neg xRy \land \neg yRx \)), then the following holds. If \( '<' \) on \( X \) is a weak order, being asymmetric and negatively transitive, then

- exactly one of \( x<'y, y<'x, x=y \) holds for each \( x,y \) in \( X \)
- \( '<' \) is transitive
- \( '=' \) is an equivalence relation
• \((x < y, y = z) \Rightarrow x < z\) and also \((x = y, y < z) \Rightarrow x < z\)

• if we define \(\prec\) on \(X/=\) (equivalence classes of \(X\) under \(=\)) as \(a \prec b \iff \exists x \in a. \exists y \in b. (x < y)\), then

• \(\prec\) on \(X/=\) is a strict order

The last statement means that is some entity in an equivalence class is less preferred than another entity in another equivalence class, then the set of equivalence classes can be put in a strict order by \(\prec\). That means for each distinct pair of equivalence classes \(a\) and \(b\), either \(a \prec b\) or vice versa.

A well-known theorem (page 14 [62]) states "if \(\prec\) on \(X\) is a weak order and \(X/=\) is countable, then there is a real-valued function \(U\) on \(X\) such that \(x < y \iff U(x) < U(y)\) for all \(x, y\) in \(X\). \(U\) is the utility measure which we propose to use as the foundation for decisions regarding schema change. Thus the existence of a utility function is assured provided we feel that the restrictions on schema are justifiable from a practical perspective.

### B.1.1 Combination of Preferences and Utility

There remains the question regarding the nature of this function which would yield a convenient measure as discussed above. This is a non-trivial task; we only enumerate the issues for completeness. Under certain conditions, \(\prec\) on \(X\) can lead to additive utility representations such as in the case of weak orders:

\[ x < y \iff u_1(x_1) + \ldots + u_n(x_n) < u_1(y_1) + \ldots + u_n(y_n), \text{ if } x = (x_1, \ldots, x_n) \text{ and } y = (y_1, \ldots, y_n) \]

If this state of affairs is valid, then we can define functions \(f\) and \(u\) additively. Such additive utilities presuppose some kinds of independence among the factors, so that the combinations of factors do not affect the whole utility. Independence assumptions mean:

\[
\begin{align*}
(x_1, x_2) &< (y_1, x_2) \Rightarrow (x_1, y_2) < (y_1, y_2) ; \text{ and} \\
(x_1, x_2) &< (x_1, y_2) \Rightarrow (y_1, x_2) < (y_1, y_2)
\end{align*}
\]

The above independence conditions are not guaranteed in the case that any one of the utility functions is a complex one of the kind \(U(x_1, x_2) = x_1 x_2 + x_1^2\). (It can be proved that no additive representation can exist if the overall utility is defined in a fashion similar to this.)

Another factor which is often involved in calculating utilities is probabilities and expected utilities. A simple probability measure on \(X\) is a real-valued function \(P\) defined on the set of all subsets of \(X\) such that:
1. $P(A) \geq 0$ for every $A \subseteq X$

2. $P(X) = 1$

3. $P(A \cup B) = P(A) + P(B)$ if $A, B \subseteq X$ and $A \cap B = \varnothing$.

4. $P(A) = 1$ for some finite subset of $X$

(3) is the finite additivity property, which is not easy to decide in the case of schema, because stating the independence of one set of queries from another is not simple, given the alternative query evaluation strategies that may be adopted depending on the data distribution at the time of the query.

If $P$ is a simple probability measure on $X$ and $f$ is a real-valued function on $X$, then the expected value of $f$ with respect to $P$ is $E(f, P) = \sum_{x \in X} f(x)P(x)$. In the context of utility of schemas, such function $E$ would need to be evaluated for every alternative schema, as a basis for deciding which alternative bears the maximum expected utility for the given set of queries. In these calculations, $P$ would serve as the a priori estimate of the percentage of certain queries against the schema, and $f$ would serve the function of the cost of that particular type of query against the schema.

B.2 Utility and Truth

According to Rescher [132], any coherent theory has truth factors associated with it, as well as use-factors (utility), which determine the value of the truth. Symbolically,

$u(S, P)$ - denotes that the situation $S$ satisfies the use-conditions of the claim $P$

$t(S, P)$ - denotes that the situation $S$ satisfies the truth-conditions of the claim $P$

Usually, $u(S, P) \rightarrow t(S, P)$, or $t(S, P) \rightarrow u(S, P)$. However, implicationally, we say *if u(S, P) and S, then !P*. In effect, this reverses the implication on practical grounds. However, we are not committed to saying $\forall$ $S$ ($u(S, P) = t(S, P)$). In other words, we are not always justified in considering all true things to be true for our practical purposes. We can reasonably stand by the statements $P \leftrightarrow t(S, P)$ and $P \leftrightarrow u(S, P)$, but the difference is that if $P \Rightarrow Q$, then $P \leftrightarrow t(S, Q)$, but not so in the case of use-conditions. So, there is a real epistemic gap between $u$ and $t$. This has to do with subjectiveness (how things stand with *us the observers* as opposed to how things stand). The usefulness of a representation (a set of beliefs regarding how the world looks) may then be decided in a background theory, as suggested by Quine’s analysis.
Appendix C: Augmented Transition Networks

C.1 The Basic ATN Formalism

Augmented transition networks (ATNs), which originated in natural language analysis, consist of labelled nodes (alternately, states) connected by directed labelled arcs (alternately, links). In natural language ATN grammars, arcs were meant to be labelled by words, syntactic categories (such as 'noun'), or the names of other ATNs. ATNs originate in transition networks, and evolve through recursive flavors of transition networks to the augmented form which is most useful for natural language processing, and under a different interpretation, for dynamic systems modeling. (See [172],[13], [25] for examples of the use of ATNs.)

Transition networks (TNs) consist of labelled nodes connected by directed labelled arcs, where the labels comprise words and syntactic categories. Recursive transition networks (RTNs) extend the formalism by allowing link labels to name other transition networks. The transition network named in the label of a particular link may be the very same network within which the link occurs, or may indirectly lead to the very same network. This adds the power of recursion to the transition network formalism. We illustrate, first, the basic traversal mechanism for RTNs.

The basic method of traversal for an RTN-specified grammar consists of a token which moves from node to node along the labeled arc connecting the nodes. If the grammar is used as a sentence generator, the actions associated with such a traversal are as follows:

- If the arc is labeled by a word, the word is output in the course of the traversal
- If the traversed arc is labeled by a syntactic category, then a word belonging to the syntactic category is output, after choosing among the words in the category.
- If the traversed arc is labelled by the name of another ATN, then that ATN is traversed (and so on, recursively) in the course of the traversal of the arc.

Notation: We use the terms Label(Node) and ATN(Label) to refer to the label of a node and the ATN named by a label respectively.

FIGURE C.1 illustrates the basic parsing mechanism. It is easy to see that RTNs essentially denote grammars, with a distinct notation. Analogously to grammars, then, RTNs may
be used to parse sentences just as they are usable in generating them. We do not detail the process of parsing using grammars; instead, we refer the reader to [13][172].

FIGURE C.1 ATN Used In Natural Language Generation / Parsing

The RTN in the figure above could be used to recognize sentences such as "John rolls a cigarette". It could just as well generate or recognize "cigarette rolls an John", a sentence which is lexically wrong and semantically suspect. In order to restrict the language generated or accepted by natural language grammars, RTNs require devices to store the necessary semantic and pragmatic information which can be used to augment their ability to accept correct sentences.

- **registers** which can store values on a global basis.
- **pre-conditions** associated with the links, which must be satisfied before a link is considered to have been traversed.
- **post-conditions** specifying constraints which must remain satisfied after the link is traversed; and
- **actions**, which can act to modify the values stored in the registers.

RTNs enhanced by these features are called ATNs. A link in an ATN is a 6-tuple:

\[ < \text{outnode}, \text{innode}, \text{label}, \text{pre-cond}, \text{post-cond}, \text{actions} > \], where
• **outnode** is a node which is the source of the link,

• **innode** is a node which is the target of the link,

• **label** is the label of the link,

• **pre-cond** is a (set of) condition(s) to be satisfied before the link is traversed,

• **post-cond** is a (set of) condition(s) referring to the registers associated with nodes of the ATN. They specify integrity conditions to hold after the traversal, and

• **actions** are \(<\text{reg}, \text{value}>\) pairs, where value may be a function of register values.

Notation: We use **Pre_Conds(Arc)**, **Post_Conds(Arc)** and **Actions(Arc)** to refer to the conjunctive pre- and post-conditions, and the actions associated with an arc, respectively. **Label(Arc)** refers to the label on an arc. **Inlinks(Node)** and **Outlinks(Node)** refer to the arcs incident on or exiting a node, respectively.

**Definition C.1** An ATN is a connected directed flow network **Graph(Nodes, Arcs)** with a distinguished Source node such that \(|\text{Inlinks(Source)}| = 0\), and a distinguished Sink node such that \(|\text{Outlinks(Sink)}| = 0\), and such that for all other nodes, \(|\text{Inlinks(Node)}| > 0\) and \(|\text{Outlinks(Node)}| > 0\). Every node and every arc has a label. Pre-conditions, post-conditions and actions are associated with each arc. By convention, an ATN is named by the label of its source node.

For convenience, we sometimes refer to a set of ATNs as an ATN itself. Each connected component is then referred to as an **ATN subnetwork**. As mentioned earlier, links in an ATN subnetwork may be labeled with the name of another subnetwork. The terminology related to this type of relationship among subnetworks is as follows:

**Definition C.2** If an ATN subnetwork A contains a link which is labeled by the name of a different subnetwork B, then A is a containing ATN of B, and B is a contained ATN within A.

Obviously, the containment relationship among ATN subnetworks may be reflexive (A can contain A) and/or symmetric (A and B may contain each other). Containment is always transitive. Direct and indirect containment follow intuitively.

The correspondence of these ATN features with inherited and synthesized attributes of attribute grammars is obvious (See [79] for a theoretical summary of grammars and languages including annotations, attribute grammars are introduced later in this appendix). In analogy with attribute grammars, then, the ATN formalism can generate context sensitive sets [98]. This is illustrated through **FIGURE C.2**, which depicts an ATN as a sentence generator (we omit the part of the ATN before the A node). For simplicity in this example, we
assume that all registers are downloaded propagated and uploaded and have the same name in all ATNs (i.e. all registers are globally visible). This figure shows that through the device of registers, an additive function and an equality check, strings of the form "a^n.b^n.c^n" may be generated. The example demonstrates the applicability of ATNs to the generation and analysis of histories including context-free and context-sensitive characteristics.

\[
\text{postcondition: } \text{len} = \text{lenc}
\]

\[
\text{current output} = a^n.b^n
\]

\[
\text{lenc} = \text{lenc} + 1
\]

\[
\text{output} = "c"
\]

FIGURE C.2 ATNs in Context-Sensitive Applications

C.2 Expressiveness Arguments for ATNs as D’foundation

In this section, we introduce the expressive power of d’foundation broadly in theoretical terms. In the thesis, we address the expressiveness of ATNs in terms which are relevant to dynamic data modeling. ATNs cover the linguistic space between and including Su’s extremes [154] due to their relationship with attribute grammars.

C.2.1 Position of ATNs in the Formal Computational Hierarchy

As a brief review, context-free grammars (CFGs) and context-sensitive grammars (CSGs) are non-length-reducing grammars, whose productions are of the form

\[
a^*Ab^* \rightarrow c^+
\]

where \( a, b, c \in N \cup T \), and \( A \in N \), and \( |c^+| > |a^*Ab^*| \) (\( N \) is the set of nonterminal grammar symbols and \( T \) is the set of terminal grammar symbols). For CFGs, \( a^* \) and \( b^* \) are null strings. With appropriate limitations, the expressive power of ATNs may be restricted to generate context free (CFLs) and context sensitive languages (CSLs) of different types. In the remainder of this section, we relate ATNs to language classes which fall within the expressive range of interest to dynamic and adaptive information systems, and through the
languages, to their recognizers. The intent is to justify our choice of the computational machine adapted for combined static-dynamic reasoning with d'language.

Briefly, attribute grammars are CFGs with inherited and synthesized attributes [98] attached to the symbols. A synthesized attribute a attached to a symbol A is calculated from the values of other attributes defined with B and C if $A \rightarrow B,C$ is a production in the grammar. An inherited attribute b attached to A may be inherited by B and C as well. ATN registers correspond to attributes of attribute grammars. ATN actions correspond to calculation of synthesized attributes and propagation of inherited attributes. ATN preconditions are simply the checks which may be made on attributes while parsing or generating sentences through an attribute grammar.

Restricted attribute grammars are recognizable by Auxiliary Pushdown Automata (APDAs) [79], which are indirectly equivalent to Linear Bounded Automata CSL recognizers. (The restriction implies ensuring that the attributes and conditions are evaluable by an APDA within a space bound.) Augmented Pushdown Automata (APDAs) are provably equivalent to Indexed Language Recognizers. Indexed Languages are Context Sensitive. The figure below illustrates the containment hierarchy.

**Unrestricted Languages**

**Context Sensitive Languages**

Indexed Languages = APDA = ATN (Restricted Space Complexity)

Context-Free Languages

Regular

Shaded area of interest to dynamic domain models

**FIGURE C.3** ATNs in the Context Sensitive Expressive Space
Appendix D: Structures in Computational Geometry

D.1 Data Structures in Computational Geometry

Temporally-dependent information in our framework is associated with half-open line segments denoting time intervals. Queries which involve the time dimension can be viewed geometrically in terms of queries over stored segments. Our data structures for such queries are based on the established theory of computational geometry data structures. Geometric queries in this context are broadly classifiable as range queries or segment queries. Range queries ("which points lie in a particular geometric region?") include orthogonal range queries, semi-dynamic range queries and non-orthogonal queries. A brief review of some relevant topics from computational geometry follows.

D.1.1 Range Queries

Orthogonal range queries return the points which are contained in hyper-rectangles (shapes whose sides are perpendicular to the co-ordinate axes). In designing our data structures, we currently adopt the orthogonality assumption, in order to simplify our problem; i.e. we assume that classes are defined geometrically by hyper-rectangles in an n-dimensional space specified by n attribute axes (This assumption can be relaxed using geometric data structures for non-orthogonal range queries, at additional cost). This limitation is justified both by the greater cost of non-orthogonal queries, and by our observation of the kinds of class definitions which are most commonly observed in dynamic and adaptive information systems.

The class definitions that are commonly encountered, particularly in schema evolution, are unions of disjoint or overlapping orthogonal hyper-rectangles. Our search space, then, contains segments within hyper-rectangles, as in figure FIGURE D.1 where the segments represent temporal intervals during which specific entities held certain values for certain (sets of) attributes. Bentley et. al [18] is an early review of data structures and search algorithms for this problem. For the basic problem of orthogonal range searching, it represents a reasonable survey to this day (see also [19], [163]). Of the surveyed algorithms for rectangle range containment (see [20] for instance), sequential scan and projection are not viable in dynamic information systems because of poor time complexity and implicit assumptions about the relative
importance of certain attributes. Of the data structures surveyed, k-ranges and range trees are similarly rejected from contention because of the $O(N \log N)$ storage requirements, where $N$ is the number of attributes to be stored.

FIGURE D.1 Range Queries

The two structures which we judge to be reasonable from the perspective of $O(N)$ space complexity are the cell structure and $k$-dimensional tree. Cell-structures are useful if the query rectangles (the $n$-hypergons within which query objects are sought) are of the same size as the cells. If this is true, as in the case of certain planning applications, cells may be the optimal data structure. In the general case when this assumption is not valid, $k$-dimensional trees are the best choice of basis for our conceptual data structures. Extending the domain to non-orthogonal queries, cutting trees of $O(\log(n))$ time, and $O(n^2)$ space complexity are useful candidates for range queries, but this extension is outside the scope of this thesis.

**D.1.2 Segment Queries**

Apart from searching for points and segments within $n$-hypergons, the other aspect of temporal queries concerns overlaps and coverages of time intervals. These are segment queries in terms of computational geometry. Segment storage is crucial for dynamic and adaptive databases because they deal with time segments during which specific attributes (or set of attributes) have values obeying certain conditions.

van Kreveld [167] summarizes data structures for geometric segment queries and gives results for hybrid data structures for combined range and segment trees. It also reviews a brute-force dynamization technique for static data structures, which we regard to be appropriate for object migration applications. Efficient data structures for segment queries are built around either the segment tree or the interval tree ([109] etc.).

**D.1.3 Segment Trees**

In a segment tree, each node represents an interval, and it's children represent intervals
which add up to the parent’s interval. Each segment (temporal interval in our case) is thus stored as a sum of its parts. It is easily derived that at each level of the binary segment tree (i.e. equidistant from the root), a particular segment may be stored at most twice. This implies a storage cost of $O(n \log N)$ for $n$ segments if the number of leaves is $N$. The storage costs of any secondary structure associated with a node is added to this. This cost yields an $O(\log N)$ query cost (plus cost of querying any associated secondary structure), again, through an easily derived result. The figure FIGURE D.2 illustrates the concept. Segment "a" is represented as a sum of the four segments marked by "*"s in the segment tree, i.e. "a" covers the starred segments.

FIGURE D.2 Segment Trees

**D.1.4 Interval Trees**

An interval tree stores $n$ intervals in linear space, and answers queries in logarithmic time.

These figures are achieved by associating each node of the interval tree with a point (a time point, in dynamic information systems), and storing each segment exactly once, at the highest node in the interval tree whose associated point is covered by the segment. In order
to achieve the logarithmic search time, the left and right endpoints of the segments stored at each node are sorted and stored in a secondary tree structure, as illustrated in figure FIGURE D.3. The cost paid for the improved time and space complexity of the interval tree comes in the form of the inflexibility of the structure. The endpoints of each segment stored at a node are committed to a sorted tree (with $O(\log n)$ worst-case search characteristics) and may not therefore be rearranged in order to achieve any other gain. Unlike the segment tree, where, for example, the segments represented at a node can be put into a range tree with respect to their x-axis coordinates for rectangle-intersection-type queries.

The conceptual data structures for dynamic and adaptive databases combine segment and range search structures, augmenting these hybrids with other structures which enable access to extensions of classes, enable classes to be defined as the unions of subclasses or intersections of other classes, etc.

D.2 Physical / File Structure Considerations

Conceptual data structures would have to rely upon file and memory structures in any implementation. Conventional relational and pre-relational databases have found ISAM structures and clustered indices (for example) to be useful. With the recent advent of commercial object-oriented database systems (which are not extensions of relational databases), the file and physical structures are increasingly treated as extensions of the physical RAM. Examples of this approach are found in the ObjectStore and Versant OODBMSs. Objects of complex and variable structures are clustered into binary large objects (BLOBs) which are stored as is in secondary storage and accessed as extensions of the RAM. All pointers within a BLOB are treated as memory pointers. Object-oriented indices contain two-level pointers to a BLOB and within it. Relational databases offer the BLOB as a standard structure as well, in recent implementations. Though this thesis does not discuss file structures, we anticipate that BLOBs, in conjunction with read-ahead access and object-oriented indices, offer the best physical medium for the realization of our conceptual structures.

D.3 Reorganizing regions and intervals for subsequent accesses

In order to restrict the cost of updates, as well as read accesses to updated data structures, techniques derived from the union-find algorithm may be used at data access time to reduce time for subsequent accesses while paying a relatively high price for the initial access.
These techniques yield a cost of the order of the inverse of the Ackerman’s function, or effectively $O(1)$ for access. We use of these techniques on interval trees and set union structures.

**D.3.1 Merging intervals at access time**

Instead of absorbing the cost of merging intervals in an interval tree for all linked intervals at the same time, partial merges can be done at access time on individual interval sequences. The strategy is simple: place the merged interval at the highest node of the interval tree where one of the merged intervals resides. The diagram below illustrates the strategy. The first time, cost is $O(n)$ where $n$ is the number of intervals merged. For subsequent accesses, the cost drops in proportion to the inverse of the Ackerman’s function.

**CONDITIONS**
- $\text{descriptor}(b) = \text{descriptor}(c) = \text{descriptor}(d) = \text{descriptor}(a)$
- data access point = interval $b$
- traversal direction = forward

**ACTIONS**
- merge $b$, $c$ and $d$ (a done lazily) to $z$
- next($a$) = $z$, previous($z$) = $a$
- next($z$) = $e$, previous($e$) = $z$
- insert $z$ at top node (highest node of traversal)

**D.3.2 Merging containers at access time**

When subset containment is indicated by doubly-linked lists of pointers with the parent set indicated by the head of the list, access costs are minimized if the list is small. However, update costs are minimized if few pointers are updated. The compromise is to update few pointers initially, and collapse pointers lazily on access (following figure). The resulting amortized complexity is bounded by the inverse of the Ackerman’s function.

**INITIAL**
- 1 step access to class name
- “B belongs to same class as A does”
- 2 step access

**ACTIONS**
- Find class($Z$)
- class($Z$) = class($B$)
- class($A$) = Set1
- Therefore link A,B, Z directly to Set1

We omit discussion of the well-known union-find algorithms and the Ackerman’s function in this thesis.
Appendix E: CSGs and Length-Reducing Histories in D'dialect(O)

The usefulness of context-constrained and forgetful histories has been discussed in terms of formal language terminology in chapter 6, leaving open the question of their specification in d’dialect, and interpretation in ATN’s. Such histories are representable through judicious use of the devices already presented in chapter 3 and 4; they require no adjectives. Formal means to model such facilities at the user level is a topic for future research.

E.1 Length-Reducing (Forgetful) Histories

The record of an object’s membership in a class C can be replaced by the record of its membership in other classes during the period of validity of its membership in C by qualifying the reaction by during, as explained in chapter 4. We note that the reverse of this process leads to the replacement of the record of membership in a sequence of classes by the record of membership in a single summary class. In using this modeling technique, however, the analyst must remember the restrictions placed on length reducing productions:

- The right-hand side of a length-reducing production must be a single symbol, which appears on the right-hand side of exactly one production.
- A length-reducing symbol can appear on the left-hand side of only length-reducing productions.
- To preserve the locally monotonic length-reducing property, a length-reducing production must never be usable to increase the length of the history, even by reversing it.

All of these conditions can be met by modeling classes and reactions in d’dialect as shown in the figure below:

The figure shows a d’dialect description of history starting from class “Start” and progressing through classes A, B and C through D. Suppose it is the analysts intention to
collapse the record of an object's membership in classes A, B ... C in an object-state as a member of the class "Length-Reduced". A reaction qualified by during is specified from Length-Reduced to A so that when the command REVERSE Tc TO Length_Reduced is used (assuming Tc is an object-state in class C), the record of membership in A, B and C is replaced by a record of membership in Length_Reduced, as per the standard rules of during. Note that:

- Length_Reduced is not reachable through forward history generation from Start. It is a unique symbol that occurs only once as the source of a reaction.
- Length-reduced itself can be the target of another length-reducing reaction R', but should not be the source of any other reaction.
- The reaction R can not be executed in a forward direction because the function f_eval is an evaluator function from which the attribute a can not be regenerated. (thus preventing R from being used to increase the length of the history).
- Reversal from D to "Length-Reduced" can be prevented by making the reaction T irreversible.

The example demonstrates the use of existing features to implement length reducing reactions. The means above can be encapsulated in a macro to simplify design, and also automatically checked, but this is beyond the scope of this thesis.

E.2 Modeling Context-Sensitive Productions

We briefly illustrate the use of pre-conditions and length-reducing productions in specifying context sensitive productions of the form:

\[ ab \rightarrow cde \], which denotes the replacement of the record of membership in classes "a followed by b" by a record of membership in classes "c followed by d followed by e". This can be done by modeling this production as a sequence of two others:

\[ b \rightarrow c'de \], with precondition: Defn(A)(O) | P, I immediately precedes membership in B

\[ ac' \rightarrow c \]. The first production is easy to model in d' dialect and ATN terms. The preconditions must be added to ensure that a record of membership in b by itself can not trigger the production (the object O may become a member of b immediately after membership in f, instead of after a). The first symbol is replaced by c', in order to enable a length reducing reaction which replaces record of membership in a and c' by record of membership in c.

We omit the details of user-input in terms of times of reaction and other attributes as details not relevant to the subject of modeling length reducing and context-sensitive rules.
References


REFERENCES


REFERENCES


REFERENCES

[85] H.V. Jagadish and X. Qian, "Integrity Maintenance in Object-Oriented Database", Proc. of the 18th Int'l Conf. on Very Large Data Bases, Vancouver, Canada, Aug. 1992, pp 469-480.
REFERENCES


[107] V.M. Markowitz, "Representing Processes in the Extended Entity-Relationship Model", in Proc. 6th IEEE Int’l Conf. on Data Engineering, Los Angeles CA, Feb. 1990, pp.103-110.


REFERENCES


[146] A. Sheth, invited talk on Transactional Workflow Systems,10th IEEE Conf. on Data Eng., Houston, TX, Feb. 1994.


REFERENCES


