**CIFE** CENTER FOR INTEGRATED FACILITY ENGINEERING

# A Data Management Model for Change Control in Collaborative Design Environment

By

Karthik Krishnamurthy
and
Kincho Law

**CIFE Technical Report #106**
**March, 1996**

**STANFORD UNIVERSITY**

© Copyright by Karthik Krishnamurthy1996
All Rights Reserved

# SUMMARY
## CIFE TECHNICAL REPORT #106

**Title:** A Data Management Model for Change Control in Collaborative Design Environment

**Authors:** Karthik Krishnamurthy and Kincho H. Law
Dept. of Civil Engineering, Stanford University

**Publication Date:** March 1996

**Funding Sources:**

- Name of Agency: National Science Foundation, Grant No. IRI-9116646-01
- Title of Research Project: Integrated Data Exchange and Concurrent Design for Engineered Facilities

- Name of Agency: Center for Integrated Facility Engineering, Stanford University
- Title of Research Project: Version and Configuration Management of 3-D CAD Object Model

### 1. Abstract:

This report presents a data management model to support collaborative design. The model monitors independent design activities by systematically tracking component descriptions in the individual disciplines. A multidisciplinary project is then described in terms of independently evolving designs from the participating disciplines. A closely coupled three-layered framework of versions, assemblies and configurations is proposed to manage independent design entities and coordination among design participants. The model has been implemented and validated on both ORACLE relational database system and AutoCAD system.

### 2. Subject:

This report describes the results of a research project that studies the issues involved in collaborative design. Based on our observations of current design practice, we propose a management solution in terms of version and configuration control of design data. In particular, a three-layered framework of versions, assemblies and configurations that can be implemented in both relational database and CAD environments is developed. The model can support project coordination through asynchronous communication of changes among designers, as well as project monitoring through systematic tracking of evolving project descriptions.

### 3. Objectives/Benefits:

Engineering design is a complex process which involves coordination of multidisciplinary efforts. For the most part, designers from each participating discipline independently develop aspects of the project from their perspectives. Individual designs are then aggregated to describe an overall project design. Besides being multidisciplinary in nature, design is an evolutionary as well as iterative process. Each designer typically generates, in parallel, several design alternatives, some of which are incrementally refined and/or modified until a satisfactory solution is obtained. The iterative design process requires maintaining descriptions of the entity at intermediate design stages of the design process. In addition, individual designers, despite their cooperative spirit, often desire autonomy and to retain control over the information they shared with others. For one, designers from different disciplines are

affiliated with different firms (or different departments within the same firm). In practice, designers independently evaluate several design options before sharing a more persistent description with the design team. The proposed data management model is designed to support collaborative design as well as allows independent control of individual design. The model has been tested in both a database management system as well as a CAD environment. It is envisioned that the model proposed herein can significantly enhance current mutidisciplinary design practice if the features described in this report are included in future CAD systems .

## 4. Methodology:

The background and status of current research related to version and configuration management as well as collaborative design issues were extensively reviewed. In addition, current design practices were studied through a real seismic retrofit design project. The change control procedures adopted by the designers were observed. The proposed data management model was then developed, taking into consideration the issues involved in current mutlidisciplinary design practice. An application to a simplified design of a Medical Cyclotron facility has been used to illustrate the practicability of the data management model.

## 5. Results:

There are two salient features of the proposed data management model. First, a three-layered closely coupled framework of versions, assemblies and configurations is developed. Configurations are used to integrate designs from each of the participating disciplines to describe an overall project. Assemblies are used to describe designs in each discipline or complex entities. Complex entities are in turn formed by aggregating instances of primitive entities in the database. An evolving description of a primitive entity is maintained using a version hierarchy, each version in a hierarchy contains specific descriptions of instances of that entity.

The second feature introduces a new concept of equivalent operations, which provides an efficient mechanism for managing changes among versions of a primitive entity. By applying this concept, we establish a version of a primitive entity as the summary of all changes that have been made to its contents. In addition, we develop operators to store, detect and characterize changes among individual versions. The close coupling of the versions, assemblies and configurations allows the computed version changes to be recursively combined to represent changes at various levels of design abstractions.

Applying these two concepts, a theoretical data management model is developed that supports project coordination through the asynchronous communication of changes among designers, as well as project monitoring through systematic tracking of evolving project descriptions. The model is tested by prototype implementations using ORACLE, a relational database software, and AutoCAD, a graphic CAD environment.

## 6. Research Status:

The theoretical framework of versions, assemblies and configurations is completed and tested. The model is ready for implementation in CAD environments. One remain issue that needs further exploration is to implement the model in a truly distributed design environment.

# Abstract

This thesis presents a data management model to support collaborative design environments. Specifically, the proposed model describes a multidisciplinary project in terms of independently evolving designs from the participating disciplines. The model monitors independent design activities by systematically tracking component descriptions in the individual disciplines. Projects are coordinated through asynchronous communication of design changes. There are two salient features of the given model. First, we specify a three-layered closely coupled framework of versions, assemblies, and configurations. In this framework, configurations integrate designs from each of the participating disciplines to describe an overall project. Assemblies can be either *total* or *partial.* The designs in each discipline are represented as *total* assemblies; *partial* assemblies represent complex entities that can be further aggregated to describe an overall design in a given discipline. Complex entities are in turn formed by aggregating instances of primitive entities in the database. We maintain an evolving description of a primitive entity as a version hierarchy; each version in a given hierarchy contains specific descriptions of instances of that entity.

The second feature introduces a new concept, *equivalent operation*, which provides an efficient mechanism for managing changes among versions of a primitive entity. Intuitively, an *equivalent operation* is a single data operation that summarizes the effect of a sequence of changes on an instance description. By applying this concept, we establish a version of a primitive entity as the summary of all changes that have been made to its contents. A version in the data management model is thus a unit of granularity whose consistency can be evaluated. In addition, we develop operators to store, detect and characterize changes among individual versions. The close coupling of the version, assembly and configuration levels allows these computed version changes to be recursively combined to represent changes at various assembly and configuration levels.

Applying these two concepts, the model efficiently supports project coordination through

the asynchronous communication of changes among designers, as well as project monitoring through the systematic tracking of evolving project descriptions. We have implemented and tested the model in both an ORACLE relational database for alpha-numeric design data as well as an AUTOCAD environment for 3-D graphical objects.

# Acknowledgments

This report is reproduced from a doctoral dissertation by Karthik Krishnamurthy submitted to Stanford University. The doctoral committee comprised of Professors Kincho H. Law (principal advisor), Jeffrey D. Ullman and Paul Teicholz. The authors would like to thank Professor Jennifer Widom, Dr. Arthur Keller, Dr. Craig Howard, Dr. Ashish Gupta and Dr. Sanjai Tiwari for their valuable discussions and feedback throughout this project.

This research was partially sponsored by the National Science Foundation, Grant No. IR-9116646, and by the Center for Integrated Facility Engineering at Stanford University.

# Contents

**Abstract** iv

**Acknowledgments** vi

**1 Introduction** 1

1.1 Need for Change Management . . . 3

1.2 Need for a Comprehensive Data Management Model . . . 5

1.3 Survey of Related Work . . . 8

1.4 Outline of Thesis . . . 13

**2 Overview of the Data Management Model** 16

2.1 Requirements of a Data Management Model . . . 18

2.2 Version Model . . . 19

2.3 Assembly Model . . . 22

2.4 Configuration Model . . . 24

2.4.1 Overview of the Configuration Model . . . 24

2.4.2 Configuration Properties to Support Collaboration . . . 25

2.5 Application of Equivalent Operations to the Proposed Framework . . . 27

2.6 Application Example . . . 29

2.7 Summary and Discussions . . . 31

**3 Version Model** 42

3.1 Overview of the Version Model . . . 45

3.2 Basic Relational Scheme . . . 55

3.2.1 Overview of the Relational Data Model . . . 55

3.2.2 Relational Implementation Scheme . . . 56

3.2.3 Modifying a Version Description . . . . . . . . . . . . . . . . . . . . 58
3.2.4 Drawbacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.3 Theory of Equivalent Operations . . . . . . . . . . . . . . . . . . . . . . 66
3.3.1 Rules to Compute an Equivalent Operation . . . . . . . . . . . . . . 66
3.3.2 Equivalent operation of a sequence of operations . . . . . . . . . . . 67
3.3.3 Final Description of Instance . . . . . . . . . . . . . . . . . . . . . 69
3.3.4 Algebraic Laws for Precedence Relationships . . . . . . . . . . . . . 69
3.3.5 Computing Equivalent Operation for a Valid Sequence of Changes . 70
3.4 Representation Scheme Based on Storing Changes . . . . . . . . . . . . . 75
3.4.1 Basic Representation Scheme . . . . . . . . . . . . . . . . . . . . . 75
3.4.2 Describing a Version Definition . . . . . . . . . . . . . . . . . . . . 77
3.4.3 Optimization for Representation Scheme . . . . . . . . . . . . . . . 78
3.5 Application of *Equivalent Operations* to the Version Model . . . . . . . . . 82
3.5.1 Describing a Version . . . . . . . . . . . . . . . . . . . . . . . . . 83
3.5.2 Computing Changes Using a *check-out/check-back-deltas* Protocol . 83
3.5.3 Integrating Application Changes with an *Active* Version . . . . . . . 87
3.5.4 Computing changes between versions . . . . . . . . . . . . . . . . . 89
3.5.5 Removing a version . . . . . . . . . . . . . . . . . . . . . . . . . . 91
3.6 Summary and Discussions . . . . . . . . . . . . . . . . . . . . . . . . . 106

**4 Assembly and Configuration Models** **109**
4.1 Assembly Model for Design Applications . . . . . . . . . . . . . . . . . . 110
4.1.1 Overview of the Assembly Model . . . . . . . . . . . . . . . . . . . 111
4.1.2 Assembly Properties for Collaboration . . . . . . . . . . . . . . . . 114
4.1.3 Relational Representation Scheme . . . . . . . . . . . . . . . . . . 116
4.1.4 Extensions to the Version Scheme . . . . . . . . . . . . . . . . . . 118
4.1.5 Assembly State Operators . . . . . . . . . . . . . . . . . . . . . . . 122
4.1.6 Assembly Change Management . . . . . . . . . . . . . . . . . . . . 125
4.2 Configuration Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
4.2.1 Overview of the Configuration Model . . . . . . . . . . . . . . . . . 131
4.2.2 Configuration Properties for Collaboration . . . . . . . . . . . . . . 134
4.2.3 Relational Representation Scheme . . . . . . . . . . . . . . . . . . 141
4.2.4 Configuration Change Management . . . . . . . . . . . . . . . . . . 143

4.3 Summary and Discussions . . . . . . . . . . 145

**5 Change Management in a CAD Environment** **147**

5.1 Application Example . . . . . . . . . . 149

5.2 Version Model . . . . . . . . . . 149

5.2.1 Representation Scheme . . . . . . . . . . 149

5.2.2 Version Change Management . . . . . . . . . . 152

5.3 Assembly Model . . . . . . . . . . 161

5.3.1 Representation Scheme . . . . . . . . . . 162

5.3.2 Modified Implementation of Assembly State Operators . . . . . . . . 164

5.3.3 Assembly Change Management . . . . . . . . . . 166

5.4 Configuration Model . . . . . . . . . . 173

5.4.1 Representation Scheme . . . . . . . . . . 173

5.4.2 Support for Project Change Management . . . . . . . . . . 174

5.5 Summary and Conclusions . . . . . . . . . . 176

**6 Summary and Conclusions** **178**

6.1 Model as a Comprehensive Data Management Solution . . . . . . . . . . 179

6.2 Limitations and Future Work . . . . . . . . . . 181

# List of Tables


2.1 Example Configurations Created by a Structural Engineer . . . . . . . . . . 25
2.2 Specifications of States of Component Assemblies . . . . . . . . . . . . . . . 34

3.1 Sequence of Operations on Version States (Application Example) . . . . . . 47
3.2 Relational Representation of a BEAM Entity . . . . . . . . . . . . . . . . . . 62
3.3 Database Operations to Specify Version States (Application Example) . . . 63
3.4 Description of the BEAM Entity after Executing the Operations in Table 3.1 63
3.5 An `update-version` Operator to Modify a Version for Change A . . . . . . 64
3.6 Change Made to Instances Contained in an *Active* Version (m-2) . . . . . . 64
3.7 Description of the BEAM Entity After Executing the Changes in Table 3.6 . 65
3.8 Instantiation of Version m-2: `materialize`(m-2) . . . . . . . . . . . . . . . 79
3.9 Function (`get-eqchange` ($u$, $v$)) for Computing an Equivalent Operation for a Pair of Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
3.10 Example Sequence of Operators on Instances of the BEAM Entity . . . . . . 86
3.11 Example Detected *Equivalent Operations* on *Active* Version (m-2) . . . . . . 86
3.12 The (`integrate-change`) Operator to Modify a Version Description. . . . . 95
3.13 `summarize` ($u$, $v$, $w$) Function for a Sequence of Changes on an Instance . . 99
3.14 Function, `merge`(*remove-id*), to Merge Changes Between the *Removed* Version and its Children . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
3.15 Merging of Changes (Contd.) (`merge`) . . . . . . . . . . . . . . . . . . . . . 105

4.1 Specifications of States of Component Assemblies . . . . . . . . . . . . . . . 137

5.1 Example Sequence of Version Operations on the BOX Entity . . . . . . . . . 152
5.2 Example Sequence of Operators on a BOX Instance 2.78672e+06 (ENTITY-OPS-LOG List) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154

5.3 Detected Change on Box Instance 2.78672e+06 (Application Example) . . 155
5.4 Example Configurations Created by a Structural Engineer . . . . . . . . . . 174

# List of Figures


1.1 Architect Delivers Initial Floor Plan . . . . . . . . . . . . . . . . . . . . . . 3
1.2 Structural Designer Independently Develops Shear Wall-Framing System . . 3
1.3 Architect Modifies Initial Floor Plan . . . . . . . . . . . . . . . . . . . . . . 4
1.4 Architect Generates Alternative Layouts before Selecting New Floor Plan . 4
1.5 Architect Presents New Floor Plan . . . . . . . . . . . . . . . . . . . . . . 5
1.6 Building Example to Illustrate the Propagation of a Single Design Change Across Disciplines and Different Levels of Detail . . . . . . . . . . . . . . . 6

2.1 Initial Floor Plan of the Facility (Assembly aa-i0) . . . . . . . . . . . . . . 34
2.2 Example Version Hierarchy of a Box Entity . . . . . . . . . . . . . . . . . . 35
2.3 Version Model for Design Applications . . . . . . . . . . . . . . . . . . . . . 36
2.4 Component Hierarchy of Architectural Assembly aa-i0 . . . . . . . . . . . . 37
2.5 Assembly Model as a Finite State Machine . . . . . . . . . . . . . . . . . . 37
2.6 Configuration Model to Support Collaboration . . . . . . . . . . . . . . . . 38
2.7 Current Structural System (Assembly sa-f0) Based on Initial Floor Plan . . 38
2.8 *Intermediate* Configuration sc-1 to Validate Current Structural System Against Initial Floor Plan . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
2.9 Alternative Layouts Generated by the Architect to Satisfy Client's Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2.10 *Intermediate* Configuration sc-2 to Validate Current Structural System Against New Floor Plan . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
2.11 *Intermediate* Configuration sc-3 to Validate Modified Structural System Against New Floor Plan . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

3.1 Initial Version Hierarchy: Beam Entity . . . . . . . . . . . . . . . . . . . . 46
3.2 Procedure to `create` a Root Version of a Primitive Entity . . . . . . . . . . 48

3.3 Procedure to activate a Version of a Primitive Entity . . . . . . . . . . . 49
3.4 Procedure to suspend a Version of a Primitive Entity . . . . . . . . . . . 50
3.5 Procedure to declare a Version of a Primitive Entity . . . . . . . . . . . 51
3.6 Procedure to derive a New Child *Active* Version of a Previously declared Version . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
3.7 Procedure to remove an Existing Version from an Entity Derivation Hierarchy 52
3.8 Version Model as a Finite State Machine . . . . . . . . . . . . . . . . . . . 53
3.9 BEAM Entity Hierarchy after Executing Operators in Table 3.1 . . . . . . . 54
3.10 Extended Representation Scheme of the BEAM Hierarchy Shown in Figure 3.9 76
3.11 Algorithm to materialize a Version Represented by Equivalent Operations 80
3.12 Procedure to complete a Version in an Entity Derivation Hierarchy . . . . 81
3.13 Procedure to materialize a Version of a Primitive Entity Represented by the "Optimized" Scheme . . . . . . . . . . . . . . . . . . . . . . . . . . 94
3.14 Relational Description of materialized BEAM Version m-2 . . . . . . . . . 95
3.15 Algorithm to compress a Sequence of Operators on Each Modified Instance to Detect the Net Changes Made During an Application Session . . . . . . 96
3.16 Procedure to Query a Version for an Instance Logically Belonging to It: query-tuple Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
3.17 Procedure to integrate Application Changes on an Entity with its *Active* Version . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
3.18 Description of the BEAM Entity after Modifying *Active* Version m-2 . . . . 99
3.19 Algorithm to compute the Changes Between Two Versions Where One Version is an *Ancestor* of the Other . . . . . . . . . . . . . . . . . . . . . . 101
3.20 Relational Representation of Changes Between Two Versions . . . . . . . . 101
3.21 Data Operations to remove a Version from an Entity Derivation Hierarchy . 102
3.22 Reconfiguring a Version Hierarchy When removing a Version Definition . . 103

4.1 Assembly Model as a Finite State Machine . . . . . . . . . . . . . . . . . . 112
4.2 Procedure to define a New Assembly in a Particular Discipline . . . . . . . 113
4.3 Procedure to generate a New Assembly as a Child of an Existing *Defined* Assembly . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
4.4 Procedure to eliminate an Existing Assembly from a Particular Discipline 115
4.5 Assembly Model as a Finite State Machine . . . . . . . . . . . . . . . . . . 116

4.6 Version Model for Design Applications . . . . . . . . . . . . . . . . . . . . . 119
4.7 Procedure to `freeze` a Particular Version in an Entity Derivation Hierarchy 120
4.8 Procedure to `thaw` a *Frozen* Version in an Entity Derivation Hierarchy . . . 121
4.9 Procedure to `publish` a Particular Version in an Entity Derivation Hierarchy 122
4.10 Procedure to `suppress` a *Published* or *Persistent* Version in an Entity Derivation Hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
4.11 Procedure to `archive` a Particular Version in an Entity Derivation Hierarchy 124
4.12 Procedure to `freeze` a *Defined* Assembly in a Particular Discipline . . . . . 126
4.13 Procedure to `thaw` a *Frozen* Assembly in a Particular Discipline . . . . . . . 127
4.14 Procedure to `publish` a *Frozen* or *Archived* Assembly in a Particular Discipline 128
4.15 Procedure to `suppress` an Assembly with an Access Property, "publish" . . 129
4.16 Procedure to `archive` a *Frozen* or *Published* Assembly in a Particular Discipline 130
4.17 Procedure to `define` a New Configuration . . . . . . . . . . . . . . . . . . . 134
4.18 Procedure to `generate` a New Configuration from an Existing Configuration Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
4.19 Procedure to `eliminate` a *Defined* Configuration . . . . . . . . . . . . . . . 136
4.20 Configuration Model to Support Collaboration . . . . . . . . . . . . . . . . . 138
4.21 Procedure to `protect` a *Defined* Configuration . . . . . . . . . . . . . . . . 139
4.22 Procedure to `unprotect` a *Defined* Configuration . . . . . . . . . . . . . . . 140
4.23 Procedure to `grant-access` to an *Intermediate* or *Recorded* Configuration . 141
4.24 Procedure to `restrict-access` to an *Accessible* or *Landmark* Configuration 142
4.25 Procedure to `stamp` an *Intermediate* or *Accessible* Configuration in a Particular Discipline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143

5.1 Initial Floor Plan of the Facility (Assembly aa-i0) . . . . . . . . . . . . . . . 149
5.2 Example Version Hierarchy of a BOX Entity . . . . . . . . . . . . . . . . . . . 157
5.3 Final Representation of the Version Hierarchy (BOX-INDEX and BOX-ACTIVE Lists) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
5.4 Instantiation of Version b-1a3 in BOX Derivation Hierarchy (`materialize`(b-1a3)) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
5.5 Description of BOX Entity Hierarchy after Modifying *Active* Version b-1a3 . 159
5.6 Computation of Changes between Versions b-1 and b-1a3 (`inserted` Operations) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160

5.7 Computation of Changes between Versions b-1 and b-1a3 (`replace` Operations) 160
5.8 Component Hierarchy of Initial Architectural Layout (Assembly aa-i0) . . . 163
5.9 Procedure to `freeze` a *Defined* Assembly in a Particular Discipline . . . . . 168
5.10 Procedure to `publish` a *Frozen* or *Archived* Assembly . . . . . . . . . . 169
5.11 Procedure to `archive` a *Frozen* or *Published* Assembly . . . . . . . . . . . 170
5.12 New Floor Plan of the Facility (Assembly aa-i1) . . . . . . . . . . . . . . . 171
5.13 Component Hierarchy of New Architectural Layout (Assembly aa-i1) . . . . 171
5.14 Current Structural System (Assembly sa-f0) Based on Initial Floor Plan . . 172
5.15 *Intermediate* Configuration sc-2 to Validate Current Structural System Against New Floor Plan . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175

# Chapter 1

# Introduction

The engineering design process is often the result of a *multidisciplinary* collaborative effort. For the most part, designers from each of the participating disciplines independently develop aspects of a project according to their individual perspectives. These individual designs are then aggregated to describe the entire project. For example, a building includes an architect's floor plan, a structural engineer's framing system, and a mechanical engineer's ducting and piping systems, among others. Furthermore, a design in a particular discipline is further described by its own component entities. For example, a particular structural frame is described by aggregating its components, including specific beams, columns and slabs.

In addition to the multidisciplinary aspect, engineering design is an *evolutionary* process. Designers typically generate several solution alternatives in parallel, incrementally refining some of them to obtain more detailed descriptions. One of the developed solution alternatives is then selected for the project. Also designers must often modify previous changes made to a design description that were later realized to be unsatisfactory. In situations where undesirable design features are not easily corrected, designers often redesign the concerned alternative from an earlier description. Redesign efforts require maintaining intermediate descriptions of entities through various stages of its evolution. Also, to monitor a design process, designers ask questions such as: *"How have the descriptions of instances of a particular entity changed in the past two weeks ?"* This involves computing changes between intermediate descriptions of the concerned design alternative.

Complexity arises in multidisciplinary design situations because changes made in one discipline commonly impact design descriptions in other disciplines. Furthermore, these

changes are propagated through different levels of detail. This thesis presents a data management model to support collaborative design environments. Specifically, the proposed model describes a multidisciplinary project in terms of the independent evolution of designs from the participating disciplines. The model monitors independent design activities by systematically tracking component descriptions in the individual disciplines. Projects are coordinated through asynchronous communication of design changes. There are two salient features of the given model. First, we specify a three-layered closely coupled framework of *versions*, *assemblies* and *configurations*. In this framework, we maintain a primitive entity as a *version* set, where a version contains specific descriptions of instances of that entity. A primitive entity, in our model, is one that can be described independently and is arbitrarily specified for a given design situation. *Assemblies* integrate component instances to describe more complex entities, as well as designs in individual disciplines. A particular component of an assembly could be either an instance of a primitive entity or another complex entity from the same discipline. *Configurations* provide a framework to describe an overall project design that is composed of designs from the participating disciplines. Second, we introduce a concept of *equivalent operations* to summarize the effect of a sequence of changes on an instance description, and apply this concept to detect, store and manage changes among versions of a primitive entity. The close coupling of the version, assembly, and configuration levels enables changes at the assembly and configuration levels to be characterized by recursively combining changes computed at the version level.

This chapter motivates this research effort, surveys related works and outlines the overall organization of this thesis. In Section 1.1, we consider a simple facility design example to underscore the importance of managing changes in typical design situations, highlighting some critical issues addressed in this study. We monitored a facility design project [26] to understand practices currently adopted by professional designers to coordinate changes among them. While a number of strategies were used, they were mostly *ad hoc* and unreliable. Section 1.2 summarizes our observations, reinforcing the need for a comprehensive model that systematically manages project changes. Although this observed example considers a design situation in the Architecture/Engineering/Construction (AEC) domain, the issues examined are broadly relevant across most multidisciplinary collaborative environments from other domains as well. In Section 1.3, we survey some of the systems that have been proposed for supporting collaboration among designers in multidisciplinary environments. While the reviewed systems address specific coordination issues, we have not

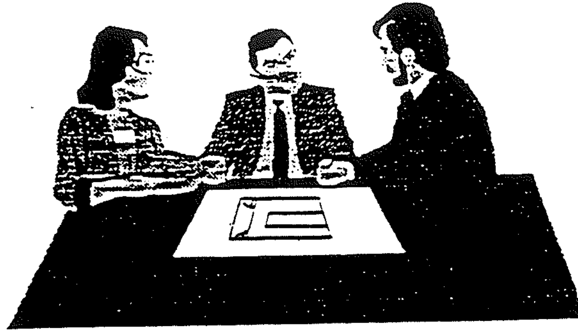


Figure 1.1: Architect Delivers Initial Floor Plan

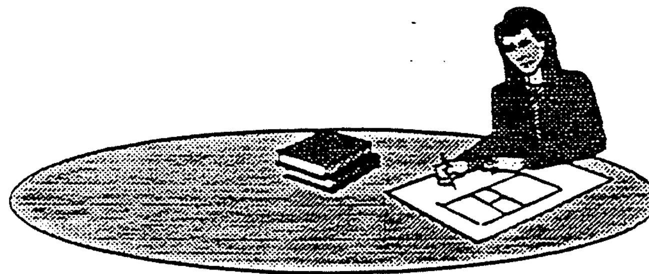


Figure 1.2: Structural Designer Independently Develops Shear Wall-Framing System

encountered any comprehensive data management model that accounts for existing design processes. We have also reviewed literature that enumerates the basic requirements for such comprehensive data management schemes. Finally, Section 1.4 outlines our discussion of the proposed model, detailing the organization of this thesis.

## 1.1 Need for Change Management

We illustrate the significance of change management in collaborative scenarios with a simple facility design example which is typical of most multidisciplinary design situations. Figure 1.1 shows a preliminary meeting at which an architect provides the team with an initial description of the floor plan. Referencing this initial layout, the structural and mechanical engineers independently design their framing and ducting systems, respectively. Figure 1.2 shows the structural engineer developing the shear wall-framing system. In the meantime, however, the architect, in consultation with the client, modifies the floor plan to better suit the facility's requirements. Figure 1.3 illustrates this situation. In fact, the architect generates several alternatives shown in Figure 1.4, before selecting one of them for the current project. Figure 1.5 fast forwards to the next team meeting at which the architect

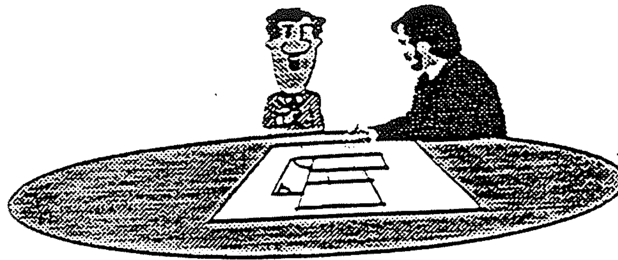


Figure 1.3: Architect Modifies Initial Floor Plan

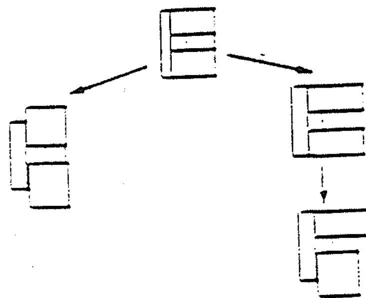


Figure 1.4: Architect Generates Alternative Layouts before Selecting New Floor Plan

presents the modified layout for the facility. Only then is the structural engineer aware that she has designed a shear wall-framing system for a floor plan that is no longer valid. To address potential inconsistencies between the existing framing system and the new floor plan, the structural engineer must know:

1. What has changed between the *new* floor plan and the floor plan that she *currently* references ?

2. Which of those changes impact her work ?

This research focuses on efficiently answering the first question, while providing support for the second.

Typically, a particular design change could impact design descriptions in several disciplines, affecting entities at various levels of detail. Figure 1.6 demonstrates how the effects of an example design change propagates. In Figure 1.6(a), the architect moves the two wall panels W1 and W2. This change makes the existing structural framing system inconsistent with the overall layout; the Y-axis frame F1 is now offset from the wall partitions. To reestablish consistency, the structural designer moves the frame F1 to coincide with the current positions of partitions W1 and W2. This is shown in Figure 1.6(b). Moving frame

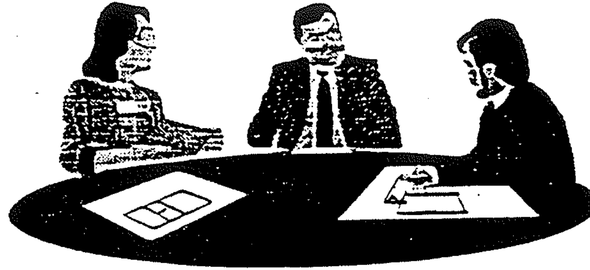


Figure 1.5: Architect Presents New Floor Plan

F1, in this fashion, increases the spans of component beams of the X-axis frames G1 and G2. Figure 1.6(c) shows the increase in the span of beam B1 in one of the X-axis frames G1. The increased beam span accounts for greater design moments in beam B1, forcing the structural engineer to redesign the beam's sectional properties (Figure 1.6(d)). This trivial example demonstrates that even a single design change (moving the wall panels in the architectural floor plan) can have a significant impact on the overall design process; the architectural change affected structural design entities at various levels of detail (from the overall 3-D framing system through the sectional properties of individual beams). Real design situations usually involve many such changes. Large projects, such as the design of an industrial plant, can have hundreds of thousands of changes propagated among the many designers who participate in the project. It is therefore obvious that change management is indeed a very complex problem.

## 1.2 Need for a Comprehensive Data Management Model

To better understand the strategies adopted by designers in professional design environments, we undertook a three month study [26] of a facility redesign project in the California Bay Area, spanning the conceptual, schematic and detailed design phases. The project involved reconfiguring a computer warehouse into a bioengineering plant. The redesigned structure was retrofitted to satisfy the latest seismic design codes. We specifically studied the processes by which designers coordinated changes in this multidisciplinary environment, identifying problems in communication and notification of design changes and other information exchanges (such as client's specifications, or rationale for design decisions among others) within the design team. In this particular project, each team member was responsible for maintaining a log of the evolution of his/her own design description, and for

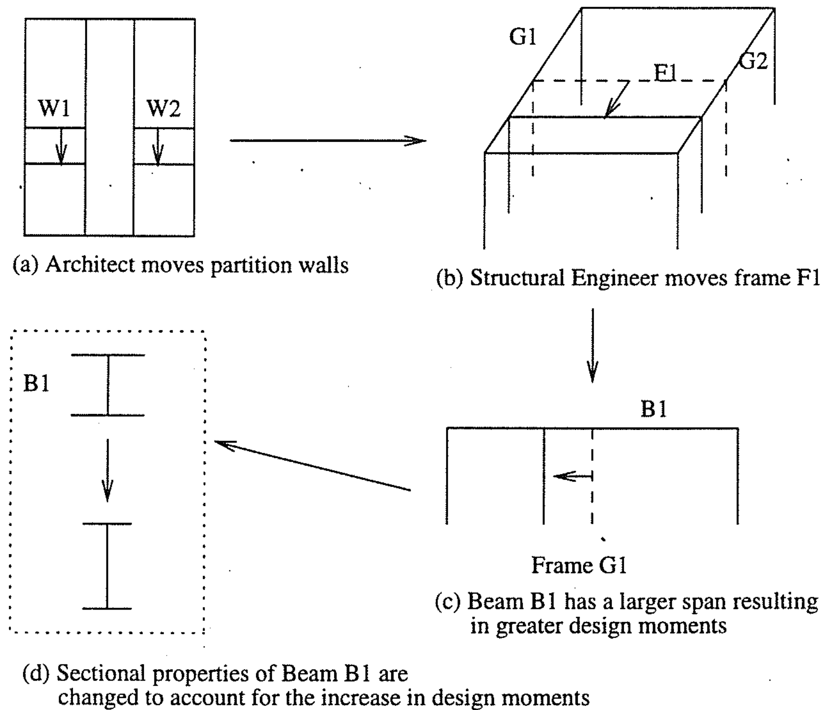



Figure 1.6: Building Example to Illustrate the Propagation of a Single Design Change Across Disciplines and Different Levels of Detail

communicating to the other team members potentially impacted by these changes. One such situation had the architect rearranging the interior spaces to meet the facility's new requirements. The architect generated an initial floor plan, and progressively modified it based on feedback from the client. Concurrency became critical as designs in the other disciplines were referencing the same floor plan even while the architect was iteratively refining it. To provide some degree of order in this dynamic environment, the architect periodically checkpointed his work, making the latest checkpointed description of the floor plan available to the other team members at weekly meetings. At each meeting, the architect would also outline how the design had changed since the previous meeting. This helped to make the other designers aware of changes that potentially impacted their work. The weekly meetings also provided a forum for designers to evaluate their individual progress, and to solicit feedback on the impact of their designs on the overall project. Outside of these weekly meetings, the designers also communicated changes, both through formal change order forms as well as through informal FAX and telephone messages.

Despite their cooperative spirit, individual designers still wished to retain control over

information they shared with others. The need for autonomy arose from a number of factors. For one, designers from different disciplines were affiliated with different organizations (or different departments within the same organization). Also, they maintained varied perspectives on the project according to their professional training. In one particular situation, we observed that the architect first independently evaluated several ceiling designs for their aesthetic appeal, before sharing possible solutions with the electrical and mechanical engineers for their feedback. This strategy gave the architect flexibility in independently evaluating several design options before sharing a more persistent description with other team members.

Unfortunately, mechanisms currently used by professional designers are usually both *ad hoc* and unreliable. Success of a specific strategy depends largely on the experience and thoroughness of the individual designers. Our study identified three categories of data management problems: (i) information transfer problems, (ii) understanding design intent problems, and (iii) storage problems. Difficulties in information transfer relate to detecting and communicating design information. In typical design situations, a number of potentially conflicting design changes go largely undetected until later in the design process. Comments such as *"Only after constructing the facility will we realize all the design inconsistencies,"* are often expressed by engineering designers. Problems related to design intent are due to insufficient understanding by individual designers of the impact of their decisions on the design processes in the other disciplines. Lastly, storage problems arise due to poor organization of design information that dynamically evolves through a design process. Such information includes both the design descriptions as well as justifications for their development.

The observations made during our study and subsequent discussions with practicing designers have convinced us that we need a comprehensive data management model for supporting collaborative design environments. The proposed model primarily addresses storage issues, describing an evolving multidisciplinary project in terms of independently evolving descriptions from the participating disciplines. The model also partially addresses information transfer issues by providing a systematic framework to store, detect, and manage changes both across disciplines and through various levels of detail in each discipline. In addition, the model supports an *asynchronous communication* scheme for coordinating changes among the design team members.

## 1.3 Survey of Related Work

Challenged by growing competitive pressures, the Architecture/Engineering/Construction (AEC) and mechanical engineering industries (among others) have been charged with looking for ways to improve their business processes by reducing costs, and by increasing quality, responsiveness, and overall consumer satisfaction. In the building industry, a series of workshops were held by the National Research Council [1, 2, 3] to develop a conceptual framework for integration of computer-based technologies in the building process. These workshops (known as Woods Hole workshops) soon focused on a conceptual framework as well as developmental needs to formulate and construct an integrated database that spans the life cycle of a facility. The integrated database (IDB) would eventually support all phases of the building project and life cycle cost considerations, providing the economic rationale for the building owner to invest in its development. The Woods Hole workshops led to the development of a prototype of an integrated database using heterogeneous hardware and software. Although the prototype was tested only for a sequential execution of typical tasks between the design and construction phases, the flexibility of the IDB to access and propagate design changes was appreciated. The core of the IDB engine, an integrator, was proposed to support the heterogeneous environment.

During the past decade, researchers have developed products and tools to support processes in design/manufacturing organizations [17]. The key challenges in implementing such Data Management Control Applications [11] has been to (i) survey current organizations, processes and procedures, then (ii) adequately defining the requirements, and finally (iii) creating the specifications necessary for evaluating available products. The existing Product Data Management systems (PDMs) have largely focused on some of the product-related activities, such as tracking design files through a company's release cycle, restricting access to such files, maintaining past versions of files for obtaining audit trails, controlling the update process, notifying users of file changes and performing electronic sign-offs. These systems have not, however, addressed capturing, managing, and coordinating processes in dynamically evolving collaborative environments.

Prasad, Morenc and Rangan [35] model concurrent environments by a number of interdependent contexts that proceed in parallel; that is, designers are often required to proceed with subjective interpretations using partial information. Based on their characterization,

Morenc and Rangan [33] have identified three requirements for concurrent engineering environments. First, concurrent activities typically involve a high degree of data and function (design) *interdependencies.* Second, in contrast to design *interdependence*, concurrency also implies the ability to design *independently.* As a result, in real design situations, a designer must make assumptions about objects for which other designers are responsible. Third, *overall consistency* must be ensured, as objects belonging to different designers evolve independently through the design process. Thus, concurrent engineering environments must be flexible enough to accommodate various strategies to meet these three requirements.

Prasad et al[35] have also outlined seven categories of information management needs of concurrent engineering environments: information modeling, teaming and sharing, planning and scheduling, networking and distribution, reasoning and negotiation, collaborative decision making and organization and management. The basis for this requirement set is detecting, managing and communicating changes within the design team, and identifying and resolving any inconsistencies among the various designs.

We have reviewed some of the prototype systems that partially satisfy the enumerated information management requirements. Initial research efforts in the computer-aided engineering community have been primarily aimed at storing multidisciplinary design data. Most of the surveyed systems use a centralized repository to maintain the final project design [38, 10]. The DICE [31] project considers a distributed environment with each agent having its own local database, although a centralized database is used to store the overall design. KADBASE [15], on the other hand, considers a heterogeneous distributed environment, addressing the issue of semantic and syntactic translations for data retrieval. However, none of these research efforts have addressed the evolution of project descriptions over time, through independent development of individual designs from the participating disciplines.

Efforts in software engineering [5, 30], engineering design [18] and document management [34] have been concerned with the problem of recording an evolving project description in terms of the independent evolution of its components. These efforts have been directed towards integrating the areas of version control and configuration management. Version control tracks the evolution of components with the duration as well as progress of the project. The goals of version control are to facilitate the efficient retrieval and storing of many versions of the same component, and to enforce restrictions for observing and controlling the evolution of each component. Configuration management, on the other hand,

is interested in putting together components to form a system. Configuring of systems can be separated into two related parts: (i) a generic description of the parts and their interrelationships that comprise the system, and (ii) an actual instantiation of a system from its generic description. It may be noted that a generic description could be represented as a product model shown by structured schemas and meta-schemas. The goals of configuration management are to facilitate the fast instantiation of a system and to enforce restrictions on the possible ways to describe a system.

The design database community has proposed versions to checkpoint evolving descriptions of a primitive entity [20, 25] and configurations to describe a composite entity in terms of specific versions of its components [20, 4]. Despite the proliferation of versioning schemes, none of the surveyed efforts have adequately addressed the interactions between versions and configurations in the context of multidisciplinary projects.

Ambriola et al[5] have traced the evolution of systems and tools used in software development environments for configuration management and version control. Their exposition is centered on two tools, the Source Code Control System (SCCS) [36] and *make* [9], which are developed through three technological generations. First generation environments, such as the Revision Control System (RCS) [40], integrated the configuration management and version control activities by gluing together the two independent tools, creating a more powerful environment for both tasks. The RCS system is at first a version control tool, enhancing the SCCS model by explicating its tree structure and increasing control by the system. Additionally, it provides a simple interface to the *make* program. A weakness of this generation of tools is that they consider their components as unstructured files; the tools fail to capitalize on situations where the contents of components are indeed restricted. The second generation systems (such as Cedar [39] and Gandalf prototype [12] systems) introduce the notion of a database; components are software objects with attributes. The Cedar system considers two concepts: configuration and description file. A configuration is an explicit description of both components and generic systems. Configurations also specify how to resolve import/export between interfaces and provide methods to perform inter-modular checking. A description file, on the other hand, contains the description of an actual file system, choosing a physical file among the possible versions for each component. In the Gandalf prototype, the organization of a system is depicted by AND/OR graphs. A module could be structured by an AND grouping of other modules. The interface for a specific module could be implemented by an OR group of possible candidates, each of which

can exist in several (OR) versions. When a system is generated, exactly one implementation and one version has to be chosen for each interface in the system. These systems do not exploit the embedded attribute schema in the database and the selection mechanisms are still rather primitive. Further, the systems assume no constructs for inter-modular type-checking, and consider no means for expressing module inter-connection. The third generation systems (such as Adele [6]) develop a language-specific system that has knowledge of the semantics of the components; this information is used to automatically construct dependency graphs among the set of components. In Adele the only requirement for the supported programming language is that components (called objects) have an interface and a body. Interfaces can exist in more versions, and bodies for each version of an interface can be in different versions or revisions. Adele introduces a family as a set of alike objects; a particular interface can be implemented by a body or another sub-family. The instantiation of a configuration is done by computing the composition list, selecting the right implementation from a user-defined description. Attributes that describe the individual objects can be used in the selection process. Version control in this system is performed on atomic objects.

Like software engineering, document management has also been an area of active research for integrating version control and configuration management activities. Peltonen et al[34] have developed an Engineering Document Management System (EDMS) that models CAD drawings and other engineering documents as objects, which can be composed of sub-documents and have multiple versions and representations. In this system, sub-documents cannot be further decomposed. Further, a document version has one version from each of its component sub-documents. A sub-document version has a primary representation and additional secondary representations. The relationships among various representations need not be symmetric. For example, a Postscript[1] representation could be generated from a CAD drawing file but not vice versa. Documents contain user defined attributes as well as actual data maintained as binary large objects (BLOBs) in a relational environment (similar to the second and third generation software development environments).

The above configuration management systems fail to meet certain crucial needs of large

[1]Postscript is a registered trademark of Adobe, Inc; *Postscript Language Reference Manual*, Addison-Wesley, 1990, second edition.

scale development efforts that involve a diverse group of professionals. Users cannot concurrently build different versions of the same system or share common derived objects. Moreover, the systems were not designed for distributed environments, which are more common in multidisciplinary situations. Leblang and Chase [30] developed a Domain Software Engineering Environment (DSEE) that addresses some of these concerns. Their system is based on the tight integration of the system builder (configuration management) and the source code control system (version control); configuration threads are a rule based description of component versions used for building a particular system. The process of instantiation evaluates the configuration thread, resolving any dynamic references to specific versions. The result is a version description containing the exact version and translation rules for each component. The cornerstone of this work is the network-wide transparent access to arbitrary versions of source elements. This assumption does not hold in usual design situations where individual designers wish to retain control over the information they share with others.

Katz et al.[20] and Chou and Kim [8] have enumerated certain interesting combinations of version properties necessary for collaboration as distinguishable version states. Similarly, Peltonen et al. have outlined document approval and release procedures by means of user-defined state graphs. Unlike the design version models of Katz and Chou, the latter document management system provides a more general mechanism for defining appropriate state graphs. This system provides authorizing and commit programs that allow greater flexibility to meet individual organization procedures. However, none of the reviewed systems have systematically identified the minimal set of basic version properties necessary to support collaboration.

Parallel research efforts have focused on coordinating concurrent changes on shared design entities. Hall [13] argues against exclusive locks when modifying design objects and proposes an asynchronous communication scheme for notifying designers of changes to objects they are currently referencing. Spooner et al.[37] have developed rules to merge concurrent changes, identifying and resolving conflicting modifications. They have not, however, addressed the integration of the merged change set with an existing version description. We are also not aware of any research efforts that address the complementary problem of detecting changes made to a structured design representation in CAD application environments.

Recently, a number of mechanisms have been proposed to support communication among

different design applications. The surveyed frameworks include blackboards [38], design critics [31], and federations of software agents coordinated by task-independent facilitators [24]. Unfortunately, these methods are fairly ad hoc in the absence of any underlying model that captures the complexity of the design process. In summary, we have encountered no comprehensive data management model that systematically and efficiently manages changes across different disciplines and through various levels of detail. For any model to be meaningful it should also parallel current practices adopted by professional designers, providing individual designers the flexibility to work independently while sharing information as necessary for cooperation. Importantly, designers need to retain control over the information for which they provide access. Development of such a model is the central focus of this dissertation.

## 1.4 Outline of Thesis

There are two salient characteristics of our proposed data management model. First, we propose a three-layered closely coupled framework of *versions*, *assemblies* and *configurations*. In this framework, configurations integrate designs from each of the participating disciplines to describe an overall project. Assemblies can be either *total* or *partial*. The designs in each discipline are represented as *total* assemblies; *partial* assemblies represent complex entities that can be further aggregated to describe an overall design in a given discipline. Complex entities are in turn formed by aggregating instances of primitive entities in the database. We maintain an evolving description of a primitive entity as a version hierarchy; each version in a given hierarchy contains specific descriptions of instances of that entity. This three-layered framework can be linked to a constraint management and notification system that represents restrictions among designs both within and across disciplines, as well as detects and notifies designers of any inconsistencies within a project description [14].

The second feature introduces a new concept, *equivalent operation*, which provides an efficient mechanism for managing changes among versions of a primitive entity. Intuitively, an *equivalent operation* is a single data operation that summarizes the effect of a sequence of changes on an instance description. By applying this concept, we establish a version of a primitive entity as the summary of all changes that have been made to its contents. A version in the data management model is thus a unit of granularity whose consistency can be evaluated. In addition, we develop operators to store, detect and characterize changes

among individual versions. The close coupling of the version, assembly and configuration levels allows these computed version changes to be recursively combined to represent changes at various assembly and configuration levels. Using these two characteristics, the model provides an infrastructure for both project coordination and monitoring.

The rest of this dissertation is organized into the following five chapters:

- Chapter 2 provides an overview of our three-layered data management model. The model supports design applications in each discipline, and provides a mechanism to integrate the individual designs to describe the overall project. We evaluate the model as a comprehensive data management solution for collaborative environments.

- Chapter 3 discusses in detail the version model to support application sessions in a particular discipline. One salient feature of the version model is the distinction made between a version's definition and its contents. A version of a primitive entity contains specific descriptions of instances of that entity. Based on the modifiability of a version's contents, we classify its into one of four states, which are sufficient to manage the design process for that entity. The version model also includes a set of basic and necessary operators to store and manage an evolving description of a primitive entity. We maintain the version set of a primitive entity as a tree structure (referred to as an entity derivation hierarchy), and develop a scheme to represent the version model in a relational environment. In this scheme, a version is maintained as a summary of all changes that were made to that version's description. *Equivalent operations* form the theoretical basis for managing changes among versions in a given entity derivation hierarchy. In this chapter, the discussion is based on the implementation of the specific version state operators as a prototype system on top of an ORACLE database system[2].

- Chapter 4 outlines the assembly and configuration models. We enumerate basic configuration properties that simulate situations realized in usual collaborative environments. Based on the assigned property values, we categorize configuration definitions into specified states. To manage the complexity of a multidisciplinary scenario, we enforce restrictions on assemblies that are included in the various configuration states. We propose assembly properties to meet these inclusion rules. Furthermore, the close

[2]ORACLE is a registered trademark of Oracle Corporation

coupling of the version and assembly models requires the properties of an assembly to be shared by each of its component versions.

- Chapter 5 describes the implementation of the data management model in a CAD paradigm. While the procedures to manage version changes are adapted from the relational paradigm, the assembly model is extended to support a recursive definition. A component of an assembly, in a CAD paradigm, could either be a primitive or a complex entity. We can recursively expand an assembly definition into a component hierarchy; the root of the hierarchy is the original assembly definition, the leaf nodes correspond to instances of primitive versions included in the assembly definition. We illustrate various aspects of the model in a CAD paradigm using an integrated example of a facility design scenario. The described facility example is simple yet realistic, and has been tested on a prototype implementation of the model in an AUTOCAD[3] environment, using AUTOLISP as the programming interface.

- Finally, Chapter 6 summarizes this work and outlines future research directions. The proposed model affords designers the flexibility to work independently while collaborating with each other on a project. We believe that the model is particularly attractive for practical implementations. This belief is grounded in our evaluation of the model as a comprehensive solution for project change control, which is independent of the underlying implementation paradigm.

[3]AUTOCAD is a registered trademark of Autodesk, Inc; *AUTOCAD Users Guide*, Release 12.

# Chapter 2

# Overview of the Data Management Model

This chapter presents an overview of a data management model for multidisciplinary design environments. Primarily, the model maintains an evolving project description in terms of independently evolving designs from each participating discipline. The model also efficiently supports **project coordination** through *asynchronous communication* of changes among designers, as well as **project monitoring** through the systematic tracking of evolving project descriptions. The basic premise is that in design situations, designers typically work independently, while sharing information necessary for collaboration. Importantly, they wish to retain control over information they make accessible to the remaining team members.

There are two salient characteristics of our proposed model. First, we propose a three-layered closely coupled framework of *versions*, *assemblies*, and *configurations*. In this framework, configurations integrate designs from each of the participating disciplines to describe an overall project. Assemblies can be either *total* or *partial*. The designs in each discipline are represented as *total* assemblies; *partial* assemblies represent complex entities that can be further aggregated to describe an overall design in a given discipline. Complex entities are in turn formed by aggregating instances of primitive entities in the database. We maintain an evolving description of a primitive entity as a version hierarchy; each version in a given hierarchy contains specific descriptions of instances of that entity. This three-layered framework can be linked to a constraint management and notification system that represents restrictions among designs both within and across disciplines, as well as detects and

notifies designers of any inconsistencies within a project description [14].

The second feature applies a new concept, *equivalent operations*, for efficiently managing changes among versions of a primitive entity. Intuitively, an *equivalent operation* is a single data operation that summarizes the effect of a sequence of changes on an instance. Using this concept, we establish a version of a primitive entity as the summary of all changes that were made to its contents. A version in the data management model is thus a unit of granularity whose consistency can be evaluated. In addition, we develop operators to store, detect and characterize changes among versions of a primitive entity. The close coupling of the version, assembly and configuration levels allows these computed version changes to be recursively combined to represent changes at the assembly and configuration levels. Using these two characteristics, the model provides an infrastructure for both project coordination and monitoring.

The purpose of this chapter is to provide an overview of the three-layered data management model and its application to collaborative design environments. Details of the version, assembly and configuration models will be discussed in subsequent chapters. The organization of this chapter is as follows: The first section outlines the requirements of a data management model to support collaborative design. These enumerated needs are guided by the design process currently practiced by the Architecture/Engineering/Construction (AEC) industry [26]. The next three sections present a brief overview of each of the version, assembly, and configuration layers of our framework, highlighting the salient features that make it attractive for practical implementations. Section 5 introduces the concept of *equivalent operations* and discusses its application to the proposed framework for supporting change management. Throughout this chapter, we illustrate various aspects of the model using an integrated example of a facility design scenario. The described facility example is simple yet realistic, and has been tested on a prototype implementation of the model in an AUTOCAD environment, using AUTOLISP as the programming interface. Section 6 demonstrates the project change management capabilities through a scripted design situation for this example facility. Finally, in the last section, we evaluate the model as a comprehensive data management solution for collaborative design environments.

## 2.1 Requirements of a Data Management Model

To better understand the collaborative design process, we undertook a three month study of a facility redesign project in the California Bay Area, spanning the conceptual, schematic, and detailed design phases. The project involved reconfiguring a computer warehouse into a bioengineering plant. The redesigned structure was retrofitted to satisfy the latest seismic design codes. We specifically focused on the processes adopted by designers to coordinate changes in this multidisciplinary environment. Based on observations made during this study and subsequent discussions with professional designers [26], we have generalized the requirements for a comprehensive data management model. The model must:

1. **Support the design of individual component entities.** The model must support the independent generation of multiple solution alternatives for the design of a particular entity. One or more of these alternatives are incrementally refined by adding more detail to their descriptions. For redesign, the model must maintain a history of previous designs; undesirable changes can be retraced to an earlier description in which they were first introduced. Monitoring the evolution of a specific alternative is then accomplished by computing the differences between descriptions of the concerned alternative over periods of time.

2. **Support the design process in an individual discipline.** The model must aggregate descriptions of individual component entities to describe complex sub-systems, as well as a complete design in a given discipline. Further, the model must trace the evolution of complex sub-designs, determining the net changes made to them over a period of time.

3. **Provide a framework to integrate designs from individual disciplines to represent a project description.** The framework must provide collaborative environments that facilitate the evaluation of project descriptions, either by individual designers or by the entire design team, collectively. Consistent project descriptions can then be shared with actors both within and outside the design team and maintained as a reference for future projects.

4. **Support the management of changes both across disciplines and through different levels of detail.** To facilitate coordination in large projects, the model must allow designers communicate their design changes with the rest of the team.

The model must also enable the project leader to monitor the overall progress of the design effort in terms of the refinements made in each of the participating disciplines.

In summary, a comprehensive data management model for collaborative design must support both the requirements of design applications in an individual discipline, as well as the integration of individual designs from the participating disciplines to describe the overall project.

## 2.2 Version Model

One salient feature of the version model is the distinction made between a version's definition and its contents. A version of a primitive entity contains specific descriptions of instances of that entity. Based on an application's ability to modify the contents of a version, we classify the version's definition into one of the following four states.

- *Active version*, whose contents are being currently manipulated by an application session. An entity can have at most one version in the *active* state.
- *Suspended version*, whose contents potentially can be modified by an application session.
- *Declared version*, whose contents can only be accessed, not altered, by an application session.
- *Removed version*, which previously existed but has since been eliminated.

These four version states are sufficient to manage the design process for an individual entity. In this model, *declared* versions correspond to checkpointed descriptions that can no longer be modified. *Suspended* versions represent solution alternatives whose descriptions can potentially be further modified. The specific alternative that is being currently designed corresponds to the *active* version.

The model includes a set of basic and necessary operators to store and manage an evolving description of a primitive entity. A version set of a primitive entity is initially specified by a `create` operation. By `activating` *suspended* versions, a designer can switch focus among several alternatives being developed in parallel. `Suspending` a version results in there being no *active* version for the given entity; a designer could temporarily divert attention to other design entities. On `declaring` a version, its description becomes checkpointed.

To further modify this alternative, a designer can `derive` a new *active* version as a child of the checkpointed description. This operation logically copies the contents of the parent into the child version; the *statically inherited* description can then be altered. A transitive closure of the parent links establishes *ancestor-descendant* relationships among versions. To control the storage needs of a version set, designers `remove` intermediate *declared* versions, as well as alternatives that are no longer being considered.

We use a hierarchical tree structure to maintain the version set of a primitive entity. We refer to the version tree as an entity derivation hierarchy. Each node in a given entity derivation hierarchy corresponds to a particular version. As the version model guarantees that a *declared* version cannot be `activated`, versions corresponding to the interior nodes of a version hierarchy are *declared.* Nodes on the fringe of a version tree, on the other hand, can be modified. A specific leaf version is *active*, while the remaining leaf nodes are *suspended.* The version model does not, however, include a `merge` operator that integrates descriptions of two parent versions into a common child. Explicitly specifying `merge` as a state operator is not justified because the detection and resolution of conflicts arising in `merge` operations depend on the particular descriptions of the two parent versions. The current versioning scheme can, however, simulate the `merging` of two parent versions by integrating the contents of one parent version into a child of the second. This results in a structure similar to a spanning version tree.

We adopt the following convention to describe versions. Version identifiers are specified as a string formed by concatenating the entity identifier and the number of the version in the derivation hierarchy. The scheme to number each version of a primitive entity is in turn adapted from [21], and implicitly stores information of the parent version in the derivation hierarchy. Thus, version b-1 is version 1 in the hierarchy of the entity "b" (Box entity in the current example).

To illustrate various aspects of the proposed model, we consider the example of a Medical Cyclotron facility built on Stanford University campus. Figure 2.1 shows a simplified initial architectural layout. To limit the scope of the example, we considered only three of the many participating disciplines: architecture, structural engineering and mechanical engineering. Figure 2.2 shows an example derivation hierarchy of a Box primitive entity. Box instances in each version of the hierarchy represent the exterior walls in the architectural layout of the Cyclotron facility. In this figure, versions b-1, b-1a0, and b-1a1 are *declared*; version b-1a2 is *suspended*, while version b-2 is *active.* Also, in this hierarchy, version b-1a0 is an

*ancestor* of version b-1a2, as it lies in the path from version b-1a2 to the root version b-1. Specific wall instances in a given version are represented by the provided scheme, and are uniquely identified by the value assigned to its primary attribute, *Box-id*. The attribute *Box-id* (generated using AUTOCAD's **handle** descriptor, "5") uniquely identifies an instance of a 3-D CAD object and is analogous to a **key** in a relational database.

To collaborate on a project, designers need to share their designs with the other team members. An access property, "publish," for a given version allows designers from other disciplines to reference its contents. Moreover, the model must ensure that a version cannot be `removed` while other designers are accessing it. We therefore specify a status property, "freeze," which restricts a version from being explicitly `removed` from an entity derivation hierarchy. Since certain versions may be components of project designs that are maintained for extended periods, often covering the facility's life cycle, they cannot be `removed`, at least for the project duration. This feature is ensured by a status property, "archive," for versions that guarantees their existence; such a version can never be `removed` from the entity hierarchy.

Based on the specific assignments of status and access property values, we further classify a version definition into one of the following four additional states[1]:

- *Frozen version,* which cannot be explicitly `removed`; status property value is "freeze." Contents of a *frozen* version are not accessible to designers from other disciplines; access value is "not publish."

- *Published version,* whose contents can be referenced by designers in other disciplines; access property value is "publish." A *published* version cannot be `removed` while being accessed by other designers; status value is "freeze."

- *Archived version,* which is guaranteed to exist for the lifetime of the database. An *archived* version is not accessible to designers from other disciplines. The version has a status value, "archive," and an access value, "not publish."

- *Persistent version*, which is guaranteed to exist for the lifetime of the database. At the same time, it is accessible to designers from other disciplines, as well as actors

[1]The *active*, *suspended*, and *declared* version states are basically concerned with managing the design process in an individual discipline and have a status property value, "not freeze," and an access property value, "not publish." It is also obvious that status and access properties are irrelevant for *removed* versions.

outside of the design team. The version has a status value, "archive," and an access value, "publish."

Figure 2.3 shows graphically the version model as a finite state machine. The various version state operators are indicated in this figure by solid arrow lines. The exception is the `derive` operator, indicated by a dashed arc, which links two versions to support the evolution of a design description. `Freezing` a version definition ensures that it cannot be `removed` without first `thawing` it. This implies that the contents of a *frozen* version are also checkpointed; a version must have been previously `declared` before it can be assigned a status property value "freeze." To ensure that a *published* version is not `removed` while it is being accessed, the model requires that only *frozen* or *archived* versions can be `published`.

## 2.3 Assembly Model

An assembly, in our model, represents a complex entity resulting from a composite modeling operation on a set of component instances. An individual component of an assembly can be an instance of either a primitive or complex entity. While the former is an instance contained in a specific version in the entity derivation hierarchy, the latter is represented by another assembly. We develop a *component hierarchy* for a given assembly by recursively expanding its definition. The root of this hierarchy is the definition of the original assembly; the leaf nodes represent primitive instances *included* in the assembly definition. Formally, an instance of a primitive entity is *included* in an assembly if it is either a component of that assembly definition or is *included* in one of its components. Furthermore, versions containing instances that are *included* in a given assembly are denoted as *included* versions.

Based on its existence, we classify an assembly definition into one of the following two states:

- *Defined assembly*, which describes an instance of a complex entity as the result of a composite modeling operation on its components.
- *Eliminated assembly*, which was previously *defined* but no longer exists.

The assembly model provides two alternative approaches to create a new *defined* assembly. A new assembly can be either `defined` independently as a composite modeling operation on its component instances, or `generated` as a *child* of another *defined* assembly. When `generating` a new assembly, a designer substitutes one or more instances *included* in the

parent assembly with more refined descriptions of those instances. The operation thus tracks an evolving description of a complex entity. Further, each *included* version in the parent assembly is guaranteed to be an *ancestor* of the corresponding *included* version (version of the same entity) in the child assembly. A transitive closure of the parent links establishes *ancestor-descendant* relationships among assemblies. An assembly is therefore a *proper ancestor* of another, if the first assembly is either (i) a parent, or (ii) a *proper ancestor* of the parent of the second assembly. Relaxing the restriction of a *proper ancestor*, we specify that an assembly is an *ancestor* of itself. The assembly model also permits the `elimination` of additional assemblies to minimize the storage requirements.

Assemblies can be alternatively classified as *total* or *partial.* A *total* assembly describes a complete design in an individual discipline and *includes* in its definition at least one instance of each entity in that discipline. A *partial* assembly, on the other hand, represents an instance of a complex entity in that discipline and can be further combined with other *partial* assemblies to describe more complex entities.

The initial architectural layout of the example Medical Cyclotron facility (shown in Figure 2.1) is represented by a *total* assembly aa-i0. Figure 2.4 shows a partial component hierarchy for this floor plan formed by the **union** of three assemblies that represent (i) its exterior walls, (ii) its interior walls, and (iii) an adjoining facility. Assembly aa-e0, which represents the set of exterior walls, is in turn formed by the **union** of individual wall assemblies. Further, a particular exterior wall assembly aa-c0 is formed by **subtracting** window and door openings (assembly aa-b0) from a solid wall. This wall corresponds to the BOX instance 2.78338e+06 in version b-1 of the entity hierarchy given earlier in Figure 2.2. Remaining elements in the example component hierarchy of assembly aa-i0 (including other exterior walls, interior walls and the adjacent facility) can be similarly expanded.

For collaborative environments, we provide status and access properties for assemblies. The possible property values and specified assembly states are similar to the version model[2]. Status and access values assigned for a particular assembly definition are shared by each of its *included* versions. This restriction is enforced by preconditions on operators that specify assembly access and status properties. Figure 2.5 shows graphically the assembly model as a finite state machine. Operators that specify various assembly states are shown in the figure as directed arcs. Such operators recursively expand an assembly definition into its

[2]Similar to the version model, the *defined* and *eliminated* states describe an assembly in the context of a single discipline. They thus have a status property value, "not freeze," and an access property value, "not publish."

component hierarchy, and check that all *included* versions as well as intermediate assemblies in the expanded hierarchy share the desired assembly properties. Pertinent state operators are invoked to assign the specified property values for components which are not already in the required state. If even one *included* version cannot be assigned the needed property value, the entire assembly state operation fails.

## 2.4 Configuration Model

We employ a configuration as a framework to represent a multidisciplinary project in terms of design descriptions from the participating disciplines. A designer in an individual discipline creates a configuration by integrating a design from his/her discipline with a design from each of the other disciplines. Complexity arises in typical multidisciplinary environments as the individual designs are not mutually exclusive; they share information such as spatial arrangements and material properties of elements in the artifact being designed. Constraints can be used to represent restrictions on an individual design due to decisions made in other disciplines [14]. As *total* assemblies represent designs in the individual disciplines, a configuration is formally specified as a set of *total* assemblies, one from each participating discipline, and a set of project constraints.

The remainder of this section is organized into two parts. First, we present an overview of the configuration model. We propose configuration states and enumerate the operators necessary to specify them. The second part describes properties that are specified for configurations to support a collaborative environment. We also present rules for assemblies (and their *included* versions) as components of a given configuration definition.

### 2.4.1 Overview of the Configuration Model

Based on its existence, we classify a configuration definition in one of the following two states:

- *Defined configuration,* which describes a project design in terms of component *total* assemblies, one from each of the participating disciplines, and an associated set of inter-disciplinary project constraints.
- *Eliminated configuration,* which was previously *defined* but no longer exists.

Table 2.1: Example Configurations Created by a Structural Engineer

| *Config* | Arch. *Assembly* | Struct. *Assembly* | Hvac *Assembly* | *Parent* |
|---|---|---|---|---|
| sc-1 | aa-i0 | sa-f0 | ha-a0 | "Null" |
| sc-2 | aa-i1 | sa-f0 | ha-a0 | sc-1 |

A configuration can be either `defined` independently, or `generated` as a child of an existing *defined* configuration. In either case, the newly created configuration is in the *defined* state. A `generate` operation substitutes one or more components of the parent configuration with more refined designs from those disciplines, thereby tracking an evolving description of a multidisciplinary project. Each component assembly in the parent configuration is guaranteed to be an *ancestor* of the corresponding component assembly (design description from the same discipline) in the child configuration. A transitive closure of parent links between configuration definitions establishes *ancestor-descendant* relationships among them. Also, an `eliminate` operation removes an existing *defined* configuration to control storage needs.

Table 2.1 shows two configurations created by a structural engineer in the example scenario. In this example, configuration sc-1 is `defined` in terms of an architectural layout (assembly aa-i0), a structural framing system (assembly sa-f0) and a mechanical ducting system (assembly ha-a0). Configuration sc-2 is `generated` as a *descendant* of configuration sc-1. The architectural layout aa-i0 in configuration sc-1 is replaced in configuration sc-2 by its *descendant* assembly, aa-i1.

### 2.4.2 Configuration Properties to Support Collaboration

Access and status properties are specified for configurations to simulate situations realized in a typical collaborative environment. Based on the specific property values assigned, a configuration can be categorized into one of the following four states[3]:

- *Intermediate configuration,* which allows a designer to privately evaluate one or more solution alternatives with respect to designs made accessible by the other disciplines. Feedback from such evaluations help designers to independently refine their individual

[3] Like the assembly model *defined* and *eliminated* configuration states have a status property value, "not freeze," and an access property value, "not publish."

designs for efficient integration with the overall project. An *intermediate* configuration cannot be accessed by other designers; access value is "not publish." In addition, such a configuration cannot be explicitly `eliminated`; the status property is assigned a value, "freeze." To satisfy the configuration properties, component assemblies from other disciplines must be at least in the *published* state, whereas the component assembly from the designer's own discipline need only be *frozen.*

- *Accessible configuration,* which simulates a meeting scenario where each designer brings his/her design to the table for the entire team to collectively evaluate the entire project description, and to identify any inconsistencies among its individual components. Such a project design can be referenced by the entire design team; the configuration has an access property value, "publish." Since the configuration can be referenced by other disciplines, it must also be guaranteed to exist (status value, "freeze"). Thus, each component assembly of an *accessible* configuration must be in the *published* or *persistent* state.

- *Landmark configuration,* which represents a consistent project description that can be shared with actors outside the design team. Such configurations represent, among others: (i) team records checkpointing descriptions of the project at the end of specific design phases, (ii) project designs submitted to regulatory agencies for construction approval, and (iii) documents released to contractors for bidding purposes. *Landmark* configurations are typically maintained for extended periods, often covering the facility's entire life cycle. Thus, such designs have a status value, "archive." Also, such configurations can be referenced by the entire team (access value, "publish"). To satisfy these properties, component assemblies of a *landmark* configuration must have a status property value, "archive," as well as an access property value, "publish." Therefore each component assembly of a *landmark* configuration must be in the *persistent* state.

- *Recorded configuration,* which represents a design containing a component solution alternative not selected for the current project, but maintained for future reference. This would allow a mechanical engineer, for example, to maintain as personal records an alternative ducting layout that was not selected for the concerned project. A *recorded* configuration has a status property value, "archive," but remains inaccessible to designers from the other disciplines. To satisfy these semantics, the assembly

included from the creator's discipline need only be *archived*, while assemblies from the other disciplines must be *persistent*.

Figure 2.6 graphically represents the configuration model as a finite state machine. In this figure, we identify the configuration states and provide operators to specify them. Table 2.2 summarizes the inclusion rules for component assemblies of the various configuration states. These proposed states represent the minimal status and access properties needed for assemblies to be components of the proposed configuration states. We enforce these inclusion rules as preconditions on operators that specify the various configuration states. The close coupling of the assembly and version models implies that the properties of a given assembly are shared by each of its *included* versions. Furthermore, we restrict certain operations that alter the states of versions and assemblies which are *included* in existing configurations. For example, a *published* assembly (or its *included* versions) cannot be `suppressed` while it is a component of either (1) an *accessible* or *landmark* configuration (having an access value, "publish"), or (2) a configuration definition created by a designer from another discipline. Similarly, a *frozen* assembly (or any of its *included* versions) cannot be `thawed` while it is a component of any existing configuration.

## 2.5 Application of Equivalent Operations to the Proposed Framework

We introduce a concept of *equivalent operations* for developing procedures to store, detect, and manage changes among versions of a primitive entity. The close coupling of the version, assembly, and configuration layers enables the management of changes at the assembly and configuration levels. We propose a *forward deltas* scheme [36] to implement the version hierarchy of a given primitive entity. Each version in this scheme contains a summary of all changes made to instances of the entity while the concerned version was *active*. Therefore, a given version can be described by executing its associated changes on the description of its parent version. *Equivalent operations* provide the theoretical foundations for operators to systematically manage changes at the version level. Intuitively, an *equivalent operation* for a sequence of changes on an instance is a single data operation that results in the same final description of the instance as the original sequence of changes. A more rigorous definition for *equivalent operations* is based on the following result:

*For any valid sequence of data operations on an instance, there exists at most one equivalent operation which is valid. This equivalent operation produces the same final description of the instance as the execution of the original sequence of data operations.*

The theoretical development of this result will be discussed in Chapter 3.

Our model adopts a *check-out/check-back-deltas* protocol to structure the interaction between an application and the *active* version of a primitive entity. At the start of a design session, a designer *checks-out* a `materialized` description of the *active* version into the CAD application. This description can then be iteratively refined using built-in CAD drawing and editing tools. An executed operation is abstracted into one of three primitive operations which capture its essence: `insert`, `delete` and `replace`. For example, a CAD operation on the geometry of an instance (scale or move) is mapped to a `replace` primitive operation. Drawing a new instance is an `insert` primitive operation, while erasing an existing instance is a `delete` primitive operation. The net changes made during a given application session correspond to the *equivalent operation* on each instance that was modified during the concerned session. At the end of a particular application session, we detect these *equivalent operations* by `compressing` the sequence of primitive operators on each modified instance. The detected changes on each entity are then *checked-back* into its *active* version; they are `integrated` with the existing description of the *active* version. As multiple CAD sessions typically *check-back* their changes to a particular *active* version, a version in an entity derivation hierarchy contains the set of *equivalent operations* of all changes on that entity that were `integrated` with the given version while it was *active*. A version is therefore described as a unit of granularity whose consistency can be evaluated. In addition, changes between an *ancestor-descendant* pair in a version hierarchy can be `computed` to effectively monitor the progress of a design process. The resulting changes represent the minimal set of data operations which can be executed on the concerned *ancestor* version to describe its *descendant*.

We can recursively aggregate determined version changes in order to represent changes among assemblies along both their *composition* and *evolutionary* relationships. While a *composition* relationship identifies instances *included* in the component hierarchy of an assembly definition, an *evolution* relationship identifies an earlier description of the complex entity from which the current assembly has been `generated`. We can therefore describe an assembly as the result of a composite modeling operation on `materialized` descriptions

of its components. Additionally, we can track the evolution of a complex entity by characterizing the changes between two assemblies in terms of the computed changes between corresponding pairs of instances *included* in both the assemblies. Characterizing assembly changes in this fashion allows individual team members to efficiently determine the changes between a recently published design and a previously published description that is being currently referenced. Such a computation of changes between *published* assemblies is, in essence, an *asynchronous communication* of design changes by a particular designer to the remaining team members, and is crucial to **coordination** in complex multidisciplinary environments.

Similarly, the version and assembly changes can be aggregated to describe changes between configurations along both the *composition* and *evolution* links. A specific configuration definition can be instantiated by aggregating descriptions of its component assemblies. The model systematically tracks an evolving project description in terms of the changes between pairs of component assemblies from each of the participating disciplines. This relieves individual designers of the burden of monitoring particular design changes, while empowering a project leader to quickly monitor the overall progress of the design process. As the computation of project changes is in terms of changes made to individual designs, a project leader can effectively ascertain the relative progress of the design process in each participating discipline. This aspect of the model can be exploited to support the accounting and scheduling of the project activities.

## 2.6 Application Example

This section previews the data management model as a comprehensive solution for project change management. We use a simple example sequence of design actions on the Cyclotron design example that was introduced in Section 2.2. This example has been tested on a prototype implementation of the model in an AUTOCAD environment [27]. Details of the individual operators as well as the implementation methodology is deferred to Chapter 5. In Chapter 5, we will also revisit the current example, though in greater detail.

First, the architect presents the initial layout of the example facility (previously given in Figure 2.1) to the remaining team members. The layout is represented as a *published total* assembly, aa-i0. Based on this layout, the structural engineer develops a shear wall-framing system represented as assembly sa-f0. Figure 2.7 shows a description of the structural design.

Concurrently, the mechanical engineer develops a ducting system for the existing layout. This ducting system is represented by a *total* assembly, ha-a0. The mechanical engineer then `publishes` the ducting system making it accessible to the rest of the team. To evaluate the consistency of the new structural design with respect to the overall project, the structural engineer `defines` a new *intermediate* configuration, sc-1, using *published* architectural and mechanical assemblies, aa-i0 and ha-a0, and the current structural design, sa-f0. Figure 2.8 shows this configuration to be consistent.

In the meantime, the architect has independently rearranged the spatial layout of the facility to suit the client's needs. In fact, multiple alternative designs were developed (Figure 2.9), one of which (assembly aa-i1) was finally selected by the client. Realizing possible effects of the layout changes on other aspects of the project, the architect `publishes` the new floor plan, aa-i1. Aware that the architect has just `published` a new floor plan, the structural engineer then reevaluates the existing shear wall-framing system against the newly `published` layout. Figure 2.10 shows the revised configuration sc-2 that is `generated` as a child of the earlier configuration sc-1. The new configuration uses the existing structural design, sa-f0, along with the previously `published` mechanical system, ha-a0, and the newly `published` floor plan, aa-i1. It then becomes clear that the existing structural design, sa-f0, is no longer consistent with the new facility layout.

To effectively address this inconsistency, the structural engineer must answer the following questions:

1. What has changed between the two architectural layouts, aa-i0 and aa-i1?
2. What is the impact of these architectural changes on the existing structural design, sa-f0 ?

By combining the `computed` changes on each component instance in the two architectural layouts, the structural engineer determines the changes between the newly `published` floor plan, aa-i1, and the previously `published` layout, aa-i0. Computation of these changes is equivalent to an *asynchronous communication* of the net changes by the architect to the structural engineer.

Given the net architectural changes, the structural engineer then redesigns the shear wall-framing system. The modified assembly, sa-f1, is again evaluated with respect to the new layout, aa-i1. Figure 2.11 shows a configuration sc-3 that is `generated` from the previous structural engineering configuration, sc-2. This configuration includes the newly

`published` architectural layout, aa-i1, the previous *published* mechanical system, ha-a0, and the newly modified structural design, sa-f1. The project design is once again consistent.

On the surface, this simple scripted example demonstrates the application of our model to possible collaborative design situations. More fundamentally, it shows an environment that provides flexibility for individual designers to work independently, while at the same time sharing information necessary for coordination.

## 2.7 Summary and Discussions

We have described a data management model to support multidisciplinary collaborative design. The model is composed of a three-layered framework of *versions, assemblies*, and *configurations.* The concept of *equivalent operations* forms the theoretical basis for operators to detect, store, and manage changes among versions in each discipline. The close coupling of the three framework layers allows the `computed` version changes to be recursively combined for describing changes at the assembly and configuration levels. Applying these concepts, the model efficiently supports **project coordination** through the *asynchronous communication* of changes among designs, as well as **project monitoring** through the systematic tracking of evolving project descriptions. The model is independent of the underlying data modeling paradigm for representing the design data. This assertion is validated in this thesis by implementing the model in both a relational (on top of an ORACLE database system using Pro *C precompiler with dynamic SQL links), and a CAD environment (in an AUTOCAD environment using AUTOLISP as the programming interface).

The proposed model parallels current design practices. The model affords designers the flexibility to work independently while collaborating with each other on a project. In this chapter, we demonstrated the project change management capabilities of our model using a simple yet realistic example of a Medical Cyclotron facility. The example has been tested on the AUTOCAD prototype implementation of our model.

We believe that this data management model is particularly attractive for practical implementation. This belief is grounded in the following special features of the model:

1. **Supports an Individual Design Process:** The version model maintains an evolving description of a primitive entity, where a version contains the set of *equivalent operations* on all instances that were modified when the given version was *active.* By

`declaring` a version, a designer checkpoints its current description for future reference. To further modify this or any other checkpointed description, the designer `derives` a new *active* version as a child of the concerned *declared* version. `Deriving` a version in this way logically copies the contents of the parent *declared* version into the new child *active* version in which it can be further modified. The model maintains the version set of a primitive entity as a hierarchical tree structure. Branching in a version hierarchy enables the parallel development of several solution alternatives; the designer switches focus to a particular alternative by `activating` it. *Suspended* versions represent alternatives that can be potentially modified upon `activation`. The `computation` of changes between two versions of an entity along the same derivation path allows a designer to monitor the design process. For redesign, a `trace` operation retraces the evolution of a design alternative, locating the *ancestor* version in which an undesirable change was first introduced. This *ancestor* version can be subsequently modified by `deriving` a new *active* child of the given version. Furthermore, designers can aggregate version changes to characterize assembly changes along both the *composition* and *evolution* links, thereby supporting the design process in an individual discipline.

2. **Supports Inter-disciplinary Collaboration:** Designers from each discipline `publish` their individual designs to share information necessary for collaboration. Importantly, the model gives designers control over the descriptions they `publish`. A configuration provides a framework to describe an entire project in terms of the component designs from each participating discipline. The project description can then be checked for overall consistency, either by individual designers or collectively by the entire design team. Configuration <u>status</u> and <u>access</u> properties simulate environments that facilitate cooperation. *Intermediate* configurations enable designers to privately evaluate one or more solution alternatives with *published* designs from the other disciplines, helping designers tailor their individual designs for efficient integration with the entire team effort. *Accessible* configurations simulate meeting scenarios in which each designer brings his/her design to the meeting table for the entire-team to collectively evaluate the project description. While *recorded* configurations represent consistent project descriptions maintained as individual records, *landmark* configurations are typically shared by actors both within and outside the design team.

3. **Supports project monitoring:** By characterizing the changes between two project descriptions in terms of the changes between their component designs, a project leader can quickly monitor the overall progress of a project in terms of the relative progress of the design process in each participating discipline. This monitoring capability can be exploited in accounting and scheduling of various design activities.

4. **Supports project coordination:** Our model does not explicitly transfer change notifications among designers. However, by `publishing` a design description, a designer shares it with the rest of the team. Individual designers can then efficiently determine the net changes between two *published* designs: one that has been newly `published`, and a previously `published` design that is being currently referenced. This is, in essence, equivalent to an *asynchronous communication* of the net design changes by an individual designer to the other team members. The key differences between our scheme and other reviewed change notification strategies are:

   (a) Only the net changes made over a period of time are communicated. This affords designers the flexibility to try various alternatives in private before making a more persistent description public.

   (b) A designer needs to provide access to a particular design before it can be referenced by other team members. This parallels current practices in professional design environments where designers typically insist on retaining control over the information they share. Although asynchronous schemes such as the one we propose can lead to delays in informing designers of a particular change, our model assumes that, in a cooperative environment, designers will be prompt in making persistent and critical changes accessible.

   (c) Presently, our model does not provide filters for individual designers to screen out those changes that do not impact them. A designer has access to the entire set of computed changes, but is left with the responsibility of identifying the smaller subset of changes that affect his/her work. Protocols to link configurations with constraint checking and notification schemes have been explored [14], but they have not been presently implemented.

In evaluating the above characteristics of our model against the set of requirements enumerated in Section 2.1, we believe that the proposed model provides a comprehensive data

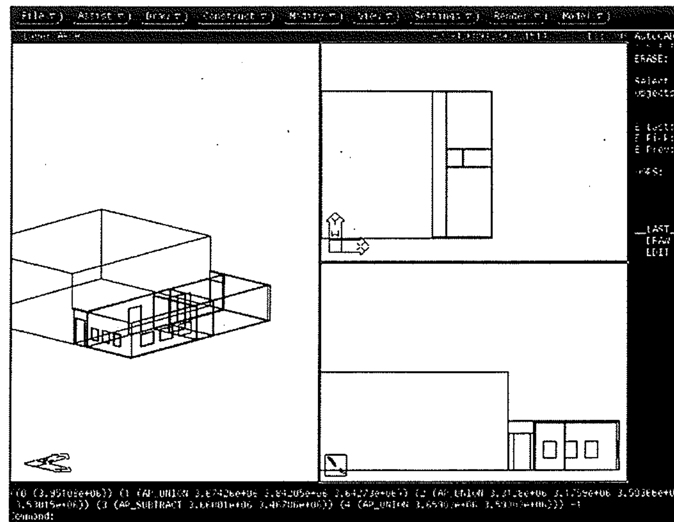


Figure 2.1: Initial Floor Plan of the Facility (Assembly aa-i0)

Table 2.2: Specifications of States of Component Assemblies

| Configuration State | Assembly State (Other Disciplines) | Assembly State (Owner's Discipline) |
|---|---|---|
| *Intermediate* | *Published* | *Frozen* |
| *Accessible* | *Published* | *Published* |
| *Recorded* | *Persistent* | *Archived* |
| *Landmark* | *Persistent* | *Persistent* |

management solution for collaborative design environments. The following two chapters describe the model in greater detail, outlining schemes to represent the model in a relational environment for alpha-numeric data. Chapter 5, on the other hand, focuses on implementing the model in a CAD environment to handle evolving graphical data.

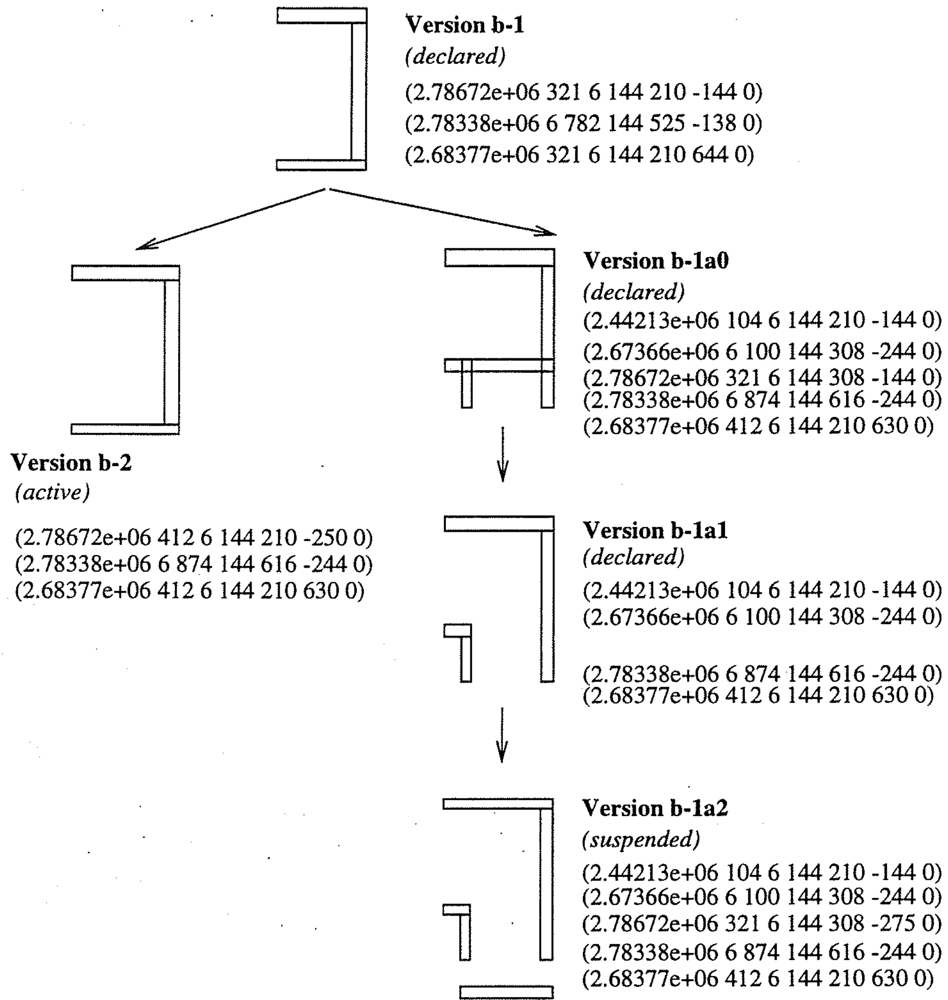



Figure 2.2: Example Version Hierarchy of a Box Entity

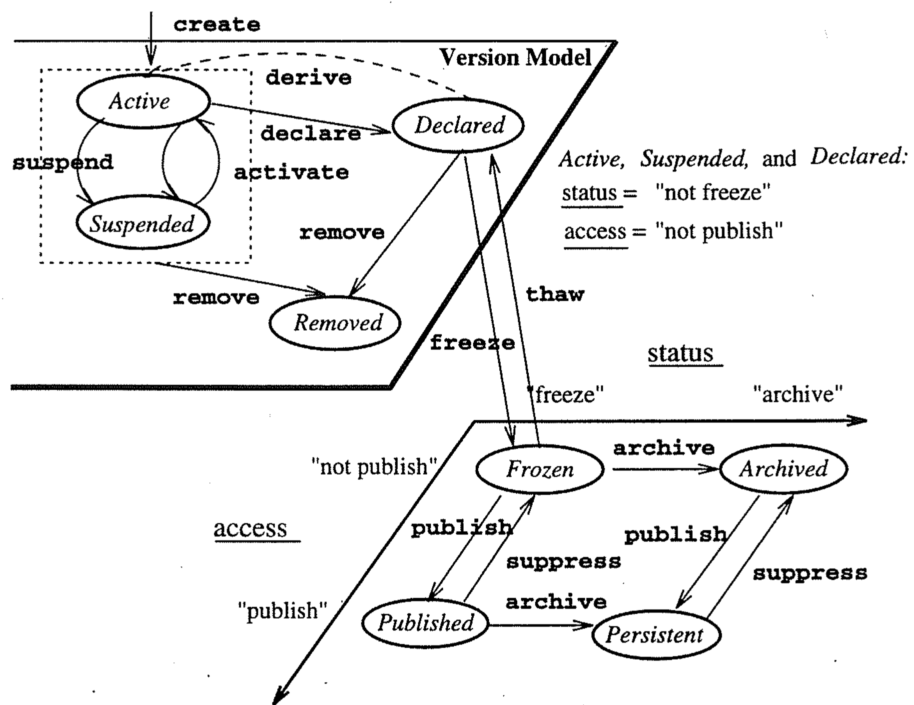



Figure 2.3: Version Model for Design Applications

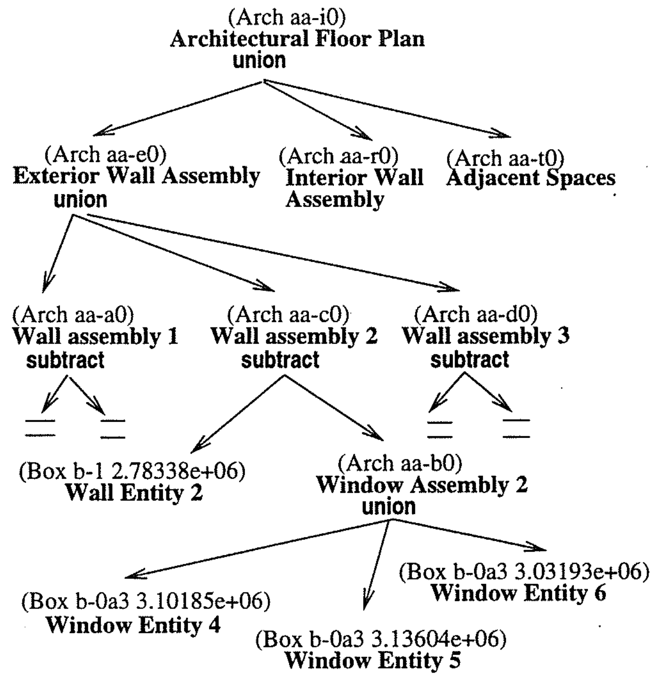



Figure 2.4: Component Hierarchy of Architectural Assembly aa-i0

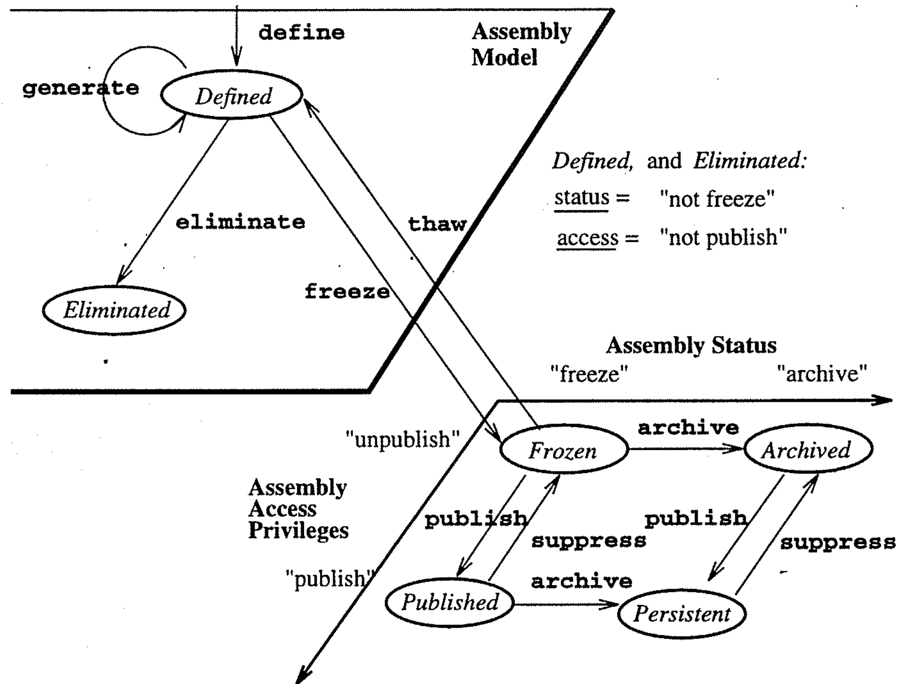



Figure 2.5: Assembly Model as a Finite State Machine

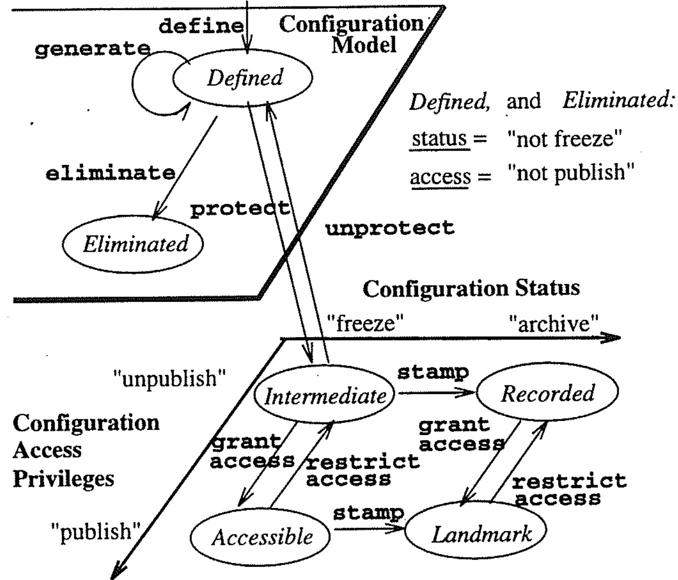



Figure 2.6: Configuration Model to Support Collaboration

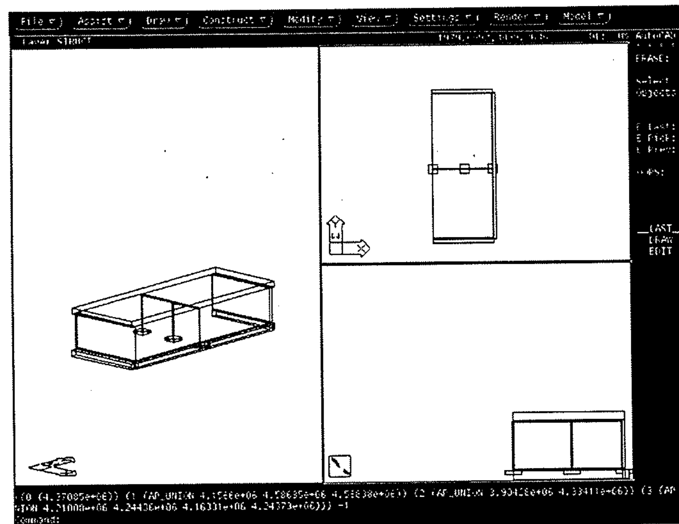


Figure 2.7: Current Structural System (Assembly sa-f0) Based on Initial Floor Plan

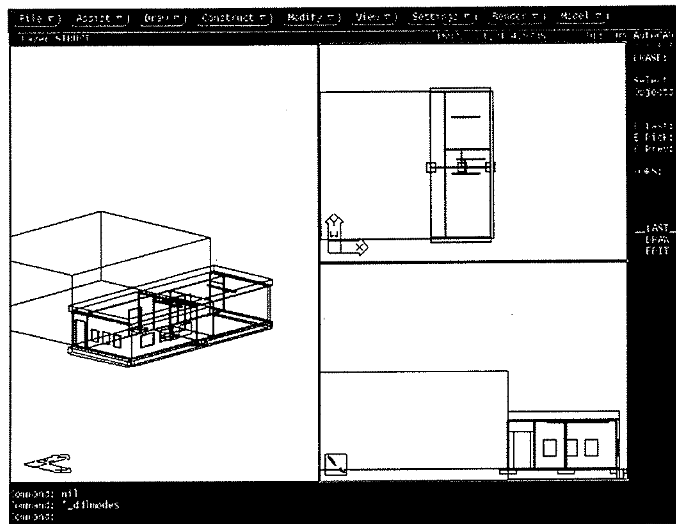



Figure 2.8: *Intermediate* Configuration sc-1 to Validate **C**urrent Structural System Against **I**nitial Floor Plan

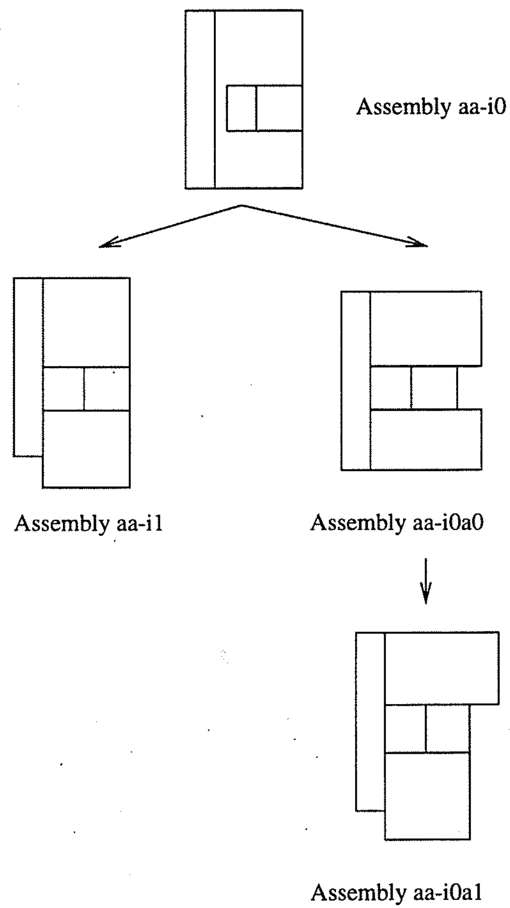



Figure 2.9: Alternative Layouts Generated by the Architect to Satisfy Client's Requirements

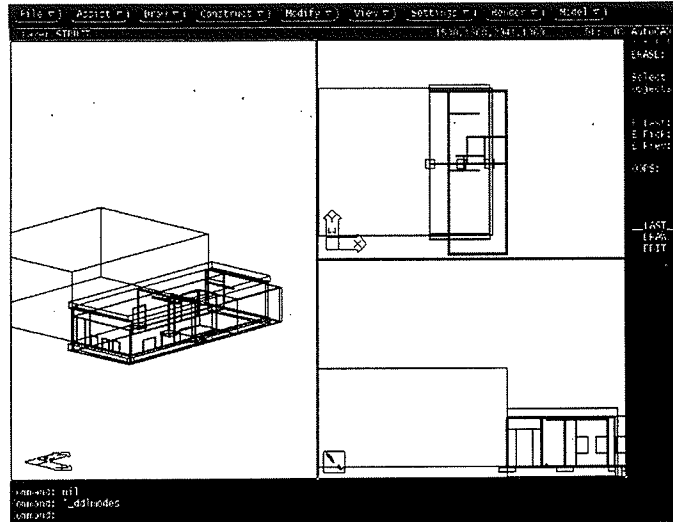


Figure 2.10: *Intermediate* Configuration sc-2 to Validate Current Structural System Against New Floor Plan

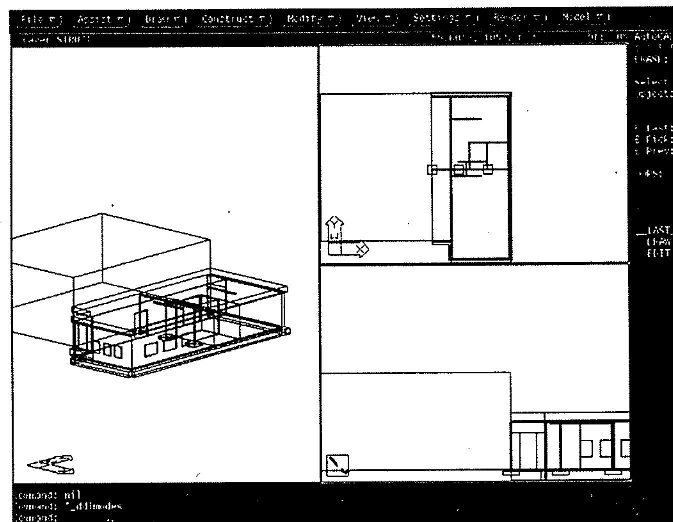


Figure 2.11: *Intermediate* Configuration sc-3 to Validate Modified Structural System Against New Floor Plan

# Chapter 3

# Version Model

This chapter presents a version model for managing descriptions of a primitive entity through its development. To design a particular entity, designers typically generate several solution alternatives, some of which are incrementally refined to obtain more detailed descriptions. In addition, designers often modify previous descriptions that are later realized as unsatisfactory. In situations where such undesirable features are not easily corrected, designers often redesign the concerned alternative from an earlier description. Redesign efforts require maintaining intermediate descriptions of entities through various stages of its evolution. Also, to monitor a design process, it is necessary to compute the changes made on instances over a period of time. By maintaining intermediate design states, designers can obtain the net changes or *deltas* more efficiently than simply tracking all design changes.

We propose a version model to maintain evolving descriptions of instances of a single primitive entity. The model maintains a version set as a tree structure, each version in a particular hierarchy contains specific descriptions of instances of that entity. Branching in a version hierarchy allows several alternatives to be developed independently. The proposed model distinguishes between the definition of a version and its contents. We, however, relate the two by classifying a version definition into four distinct states, three of which (*active*, *suspended* and *declared*) are based on the modifiability of its contents. *Declared* versions represent previously checkpointed descriptions of an entity. *Active* and *suspended* versions, on the other hand, are modifiable; the *active* version corresponds to the current focus of a design application, while *suspended* versions represent solution alternatives which could be refined by making them *active*. Finally, the fourth *removed* state is mainly for record-keeping purposes. These version states are the minimal number of states needed to support

an individual design process. We maintain the version set of a primitive entity as a tree structure, and specify operators to manage the version hierarchy for each entity. The first section of this chapter discusses, in detail, the version model for managing the design of a primitive entity.

In addition, we develop operators to store and manage changes both within and across versions. Unlike the operators on a version definition, procedures on its contents depend on the data modeling paradigm used to describe the design entities. This chapter primarily considers the implementation of the version model in a relational environment, although the algorithms developed here are applicable to other underlying data models as well. In this context, we specify a primitive entity by a relational scheme; a given instance is a tuple in that relation. A change is an `insert/delete/replace` operation on a tuple. Section 3.2 presents a basic implementation scheme that associates all instances contained in a version with a system-generated identifier. Using this scheme, the procedure to `derive` a new version physically copies all tuples from the parent to the child version. Given that we typically modify only a fraction of a version's contents while it is *active*, the present scheme would store multiple copies of the same instance description in versions along a derivation path. To overcome this obvious drawback, we apply a new concept, *equivalent operations*, to develop a more compact versioning scheme. Intuitively, an *equivalent operation* is a single data operation that summarizes the effect of a sequence of changes on an instance. Section 3.3 then establishes some fundamental properties for *equivalent operations*. By applying the developed concepts, we represent a version of a primitive entity as a set of *equivalent operations* on instances of that entity. Section 3.4 provides a compact *forward deltas* scheme [36] to represent a version hierarchy. This scheme is efficient in terms of storage; each version contains only a summary of all changes made to that entity while the given version was *active*. Tuples belonging to a version description could be either physically associated with that version itself, or *inherited* from one of its *ancestors*. We describe a particular version by retracing its derivation path till the root version, collecting from each of its *ancestors* those tuples that logically belong to its definition. Instantiating a version in this fashion can be computationally expensive in large design hierarchies. Our approach to improving computational efficiency is to explicitly store instantiated descriptions of certain intermediate versions in the version hierarchy. We call such instantiated versions as *complete*; the remaining versions are denoted as *incomplete*. By this efficiency for storage tradeoff, procedures to describe a version need to retrace the derivation path only until the most recent

*complete ancestor* version, which physically contains all tuples from its own *ancestors* that could be potentially *inherited* by the version being described.

In Section 3.5, we apply *equivalent operations* to store, detect, and manage changes among versions in an entity hierarchy. We introduce a *check-out/check-back-deltas* protocol to structure the interaction between an application and the *active* version of a primitive entity. Using this protocol, a designer initially *checks-out* a description of the *active* version into the design application, where it can be modified. At the end of the session, the designer *checks-back*, into the *active* version, the net changes made during that session. We represent the *checked-back* design changes as `insert`, `delete`, or `replace` *equivalent operations* on the modified instances. These application changes are determined by a `compress` operation. The *checked-back* changes on an entity are then merged with its *active* version. An `integrate` operator obtains the *equivalent operations* for the *checked-back* changes and the present *active* version description. The *active* version now contains the new set of *equivalent operations.* Since multiple application sessions typically *check-back* their changes while a particular version is in the *active* state, a version, in our scheme, contains the set of *equivalent operations* of all changes that were *checked-back* to that version. By maintaining versions as a set of *equivalent operations*, we can efficiently `compute` the net changes between two versions, where one is an *ancestor* of the other. These `computed` changes are the minimal set of data operations that can be executed on the *ancestor* version to describe its *descendant.*

Throughout this chapter, we illustrate various aspects of the model using an example derivation hierarchy of a single primitive entity, BEAM. This example has been tested on a prototype implementation of the model in an ORACLE[1] relational database system. The prototype has been implemented in the C programming language using dynamic SQL links that are supported by a Pro*C precompiler.

Version control has been an active research area in the software engineering, CAD and database communities. In the last section, we summarize our version model in the context of previous research developments, while highlighting our specific contributions to the field.

---

[1]ORACLE is a trademark of ORACLE Corp.

## 3.1 Overview of the Version Model

Based on its existence and the modifiability of its contents, we classify a version definition into one of the following four states:

- *Active version*, whose contents are being currently manipulated by an application session. An entity can have at most one version in the *active* state.
- *Suspended version*, whose contents can potentially be modified by an application session.
- *Declared version*, whose contents can only be accessed, but not altered, by an application session.
- *Removed version,* which previously existed, but has since been eliminated.

We use a tree structure to maintain the version set of a primitive entity, and refer to it as an entity derivation hierarchy. Each node in a given entity hierarchy corresponds to a particular version. Since a *declared* version cannot be reassigned to an *active* state, versions that correspond to interior nodes of a given version hierarchy are *declared.* On the hand, nodes on the fringe of a version tree can usually be modified; a single leaf version could be *active*; the remaining leaf nodes are likely to be *suspended.* While it is possible for a leaf node to be *declared*, an existing interior version cannot be in the *suspended* or *active* states.

The model includes a set of basic and necessary operators on a version definition to generate and maintain a version tree. A version set of a primitive entity is initially specified by a `create` operation. By `activating` *suspended* versions, a designer can switch focus among several alternatives being developed in parallel. `Suspending` a version results in there being no *active* version for the given entity; a designer could temporarily divert attention to other design entities. On `declaring` a version, its description becomes checkpointed. To continue modifying this or any other checkpointed description, a designer `derives` a new *active* version as a child of the concerned *declared* version. This operation logically copies the contents of the parent version into the child; this *statically inherited* description can now be altered. To control the storage needs of a version set, designers `remove` intermediate *declared* versions, as well as alternatives that are no longer being considered.

Throughout this thesis, we use the following notation: Version identifiers are specified as a string formed by the concatenation of the entity identifier and the number of the version

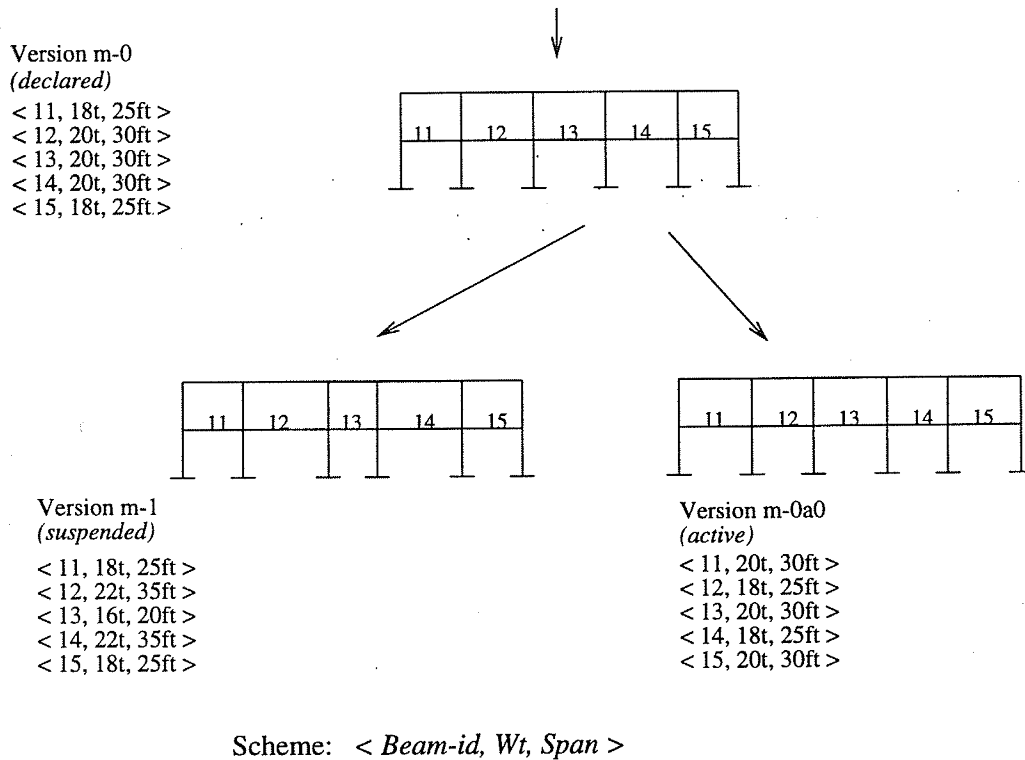



Figure 3.1: Initial Version Hierarchy: BEAM Entity

in the derivation hierarchy. The version numbering scheme is in turn adopted from [21], and implicitly stores information of the parent version in the derivation hierarchy. A version m-1 in this scheme is, thus, version 1 in the hierarchy of the entity "m" (BEAM entity in the current illustrative example).

Figure 3.1 shows an example derivation hierarchy of a BEAM primitive entity, each version in the hierarchy contains component beams of the 2-D frame. In Figure 3.1, version m-0 is *declared*; version m-1 is *suspended*, while version m-0a0 is *active*. We describe the BEAM instances in each version using the provided scheme. A primary attribute, *Beam-id*, uniquely identifies instances in a particular version. The remaining secondary attributes, *Wt* and *Span*, describe the design properties of each BEAM instance.

The rest of this section develops procedures for version state operators that generate and manage a version hierarchy. We describe these procedures using an ALGOL 68 syntax, though for clarity we also use ALGOL 60 notations. Procedures for the version state operators use the following data type declarations:

Table 3.1: Sequence of Operations on Version States (Application Example)

| Operations | Version Affected | Initial State | Final State | Version Hierarchy Changes |
|---|---|---|---|---|
| `activate`(BEAM, m-1) | m-0a0 | *active* | *suspended* | none |
| | m-1 | *suspended* | *active* | none |
| `declare`(BEAM, m-1) | m-1 | *active* | *declared* | none |
| `derive`(BEAM, m-2) | m-2 | not existent | *active* | version m-2 becomes child of version m-1 |

- entity: Name of the entity under consideration.
- $id_{version}$: Identifier of given version belonging to an entity derivation hierarchy.
- primary: Set of primary attributes that uniquely identify each instance of an entity.

In addition, the proposed version algorithms use the following *external functions*:

- `vers-numgen` = (entity E, $id_{version}$ P) $id_{version}$ V: Generates an identifier V for a new version of a primitive entity E that is `derived` as a child of previously *declared* version P. Using the adopted numbering scheme [21], we can uniquely generate an identifier for a version, given the identifier of its parent and the version's position with respect to its siblings. A `vers-numgen` function requests the user for the former value, the system internally maintains the number of children at each node. The child counter for a node is incremented each time a `vers-numgen` operator is executed at that node. When `creating` a root version of a primitive entity, the parent identifier P is assigned a value, "Null."
- `vers-parent` = (entity E, $id_{version}$ V) $id_{version}$ P: Obtains the parent P of a version V in the derivation hierarchy of entity E. As the version set is maintained as a tree structure, all versions except the root have exactly one parent.
- `vers-state` = (entity E, $id_{version}$ V) string S: Determines the current state specification S of a version V belonging to the derivation hierarchy of entity E.

We now present the six basic operators necessary to develop an entity derivation hierarchy to specify the state of each included version.

**Algorithm 1** (`create`)

*Input: e:* Name of the primitive entity whose version hierarchy is being `created`.

*Output: v:* Identifier of the root version in the derivation hierarchy of entity e.

**procedure** `create` = (entity e) $\text{id}_{\text{version}}$ v:
**begin**
  v := `vers-numgen`(e, "Null");
  `vers-state`(e, v) := *active*;
**end procedure**
◇

Figure 3.2: Procedure to `create` a Root Version of a Primitive Entity

- `create`(E): Specifies the root version of the derivation hierarchy of an entity E and determines it as *active*. A `create` operation initiates a call to a system function `vers-numgen` that generates an identifier V for the version. Figure 3.2 outlines an algorithm for the `create` operator.

- `activate`(E, V): Takes a *suspended* version V of an entity E and makes it *active*. A version that was *active* before the execution of the operator is now specified as *suspended*. By `activating` a particular design alternative, a designer changes focus to the concerned design solution from some other alternative. Therefore, an `activate` operator allows a designer to develop several alternatives in parallel, while switching his/her attention among them. A successful `activate` operation ensures that version V is in the *active* state. The operation fails only if an existing version is in the *declared* state. An algorithm to implement this operator is presented in Figure 3.3.

- `suspend`(E, V): Specifies an *active* version V of entity E as *suspended*. A successful execution of this operator results in there being no *active* version in the derivation hierarchy. By `suspending` an *active* version, a designer temporarily halts efforts directed towards entity E, diverting attention to other entities. Figure 3.4 outlines a procedure to `suspend` a version.

- `declare`(E, V): Specifies an *active* version V of entity E as *declared*. `Declaring` a version checkpoints its description for future reference. Further, a *suspended* version needs to be `activated` before it can be `declared`. An algorithm for the `declare`

**Algorithm 2** (`activate`)

*Input:* *e:* Name of the primitive entity of interest.
  *v:* Identifier of the version being `activated`.
*Output:* *success:* Boolean indicator of a successful operation.

**procedure** `activate` = (entity *e*, $\text{id}_{\text{version}}$ *v*) boolean *success*:
**begin**
  **if** ((`vers-state`(*e*, *v*) = *suspended*) OR (`vers-state`(*e*, *v*) = *active*)) **begin**
    **if** (`vers-state`(*e*, *v*) = *suspended*) **begin**
      `vers-state`(*e*, *v*) := *active*;
    **end**;
    *success* := TRUE;
  **end**;
**end procedure**
◇

Figure 3.3: Procedure to `activate` a Version of a Primitive Entity

operator is presented in Figure 3.5.

- `derive`(E, P): Creates a new *active* version V of entity E, and links it as a child of a *declared* version P. The operation invokes the system function `vers-numgen` for generating an identifier V for the new child version. Further, the new child version is specified as *active*. As a `derive` operation logically copies the contents of the parent version into the child, a designer can modify a design alternative even after checkpointing its description. The `derive` operation is fundamental to extending a version hierarchy. Figure 3.6 provides a procedure for a `derive` operation.

- `remove`(E, V): Specifies an existing version (*active*, *suspended*, or *declared*) V of entity E as *removed*. Storage considerations prompt procedures to prune version hierarchies by `removing`: (i) alternatives that are no longer part of a design solution, and (ii) certain intermediate checkpointed descriptions that were created early in the design process. Figure 3.7 shows an algorithm to `remove` a version definition from an entity hierarchy. Since a *removed* version does not exist anymore, its contents must also be erased. The procedure to erase the contents of a version being `removed` depends on the underlying data modeling paradigm, as well as the representation scheme used

**Algorithm 3** (suspend)

*Input: e:* Name of the primitive entity of interest.
*v:* Identifier of the version being `suspended`.

*Output: success:* Boolean indicator of a successful operation.

```
procedure suspend = (entity e, id_version v) boolean success:
begin
   success := FALSE;
   if ((vers-state(e, v) = suspended) OR (vers-state(e, v) = active)) begin
      if (vers-state(e, v) = active) begin
         vers-state(e, v) := suspended;
      end;
      success := TRUE;
   end;
end procedure
◇
```

Figure 3.4: Procedure to `suspend` a Version of a Primitive Entity

to implement the version model. In Section 3.5.5, we will outline a procedure for `removing` the contents of a version in the relational context.

The above set represents a minimal set of operators needed to manage a design process for a primitive entity. Figure 3.8 shows graphically the version model as a finite state machine. The various version state operators are indicated in this figure by solid arrow lines. The exception is the `derive` operator, indicated by a dashed arc, which links two versions to generate an entity derivation hierarchy. The version model does not, however, explicitly support a `merge` state operator that integrates descriptions of two parent versions into a common child. The detection and resolution of conflicts that arise in `merge` operations depend on the specific descriptions of the two parent versions, and do not justify specifying `merge` as a state operator. The current versioning scheme can however simulate the `merging` of two parent versions by integrating the contents of one of the parent versions into a child of the second. This results in a structure similar to a spanning version tree.

We demonstrate the version model, using the "ongoing" example of the BEAM entity. Specifically, we execute the following sequence of operators on the BEAM version hierarchy shown in Figure 3.1.

**Algorithm 4** (declare)

*Input:* *e:* Name of the primitive entity of interest.
 *v:* Identifier of the version being declared.

*Output:* *success:* Boolean indicator of a successful operation.

```
procedure declare = (entity e, id_version v) boolean success:
begin
   success := FALSE;
   if ((vers-state(e, v) = active) OR (vers-state(e, v) = declared)) begin
      if (vers-state(e, v) = active) begin
         vers-state(e, v) := declared;
      end;
      success := TRUE;
   end;
end procedure
◇
```

Figure 3.5: Procedure to declare a Version of a Primitive Entity

1. activate(BEAM, m-1): This operation specifies version m-1 as *active*. The version m-0a0 which was initially *active* is now specified as *suspended.*

2. declare(BEAM, m-1): This operation specifies version m-1 as *declared.*

3. derive(BEAM, m-1): This operator creates a new *active* version m-2 and links it as a child of the previously *declared* version m-1. At the time of derivation, the contents of the parent version m-1 are logically copied into the child version m-2.

The execution of the above sequence of operations is summarized in Table 3.1. Figure 3.9 shows a final description of the example BEAM derivation hierarchy.

**Algorithm 5** (`derive`)

*Input:* *e:* Name of the primitive entity of interest.
*p:* Identifier of the *declared* version, a child for which is being `derived`.

*Output:* *v:* Identifier of the new *active* version which is a child of version *p*.

**procedure** `derive` = (entity *e*, $\text{id}_{\text{version}}$ *p*) $\text{id}_{\text{version}}$ *v*:
**begin**
  **if** (`vers-state`(*e*, *p*) = *declared*) **begin**
    *v* := `vers-numgen`(*e*, *p*);
    `vers-state`(*e*, *v*) := *active*;
    `vers-parent`(*e*, *v*) := *p*;
  **end**;
**end procedure**
◇

Figure 3.6: Procedure to `derive` a New Child *Active* Version of a Previously `declared` Version

**Algorithm 6** (`remove`)

*Input:* *e:* Name of the primitive entity of interest.
*v:* Identifier of the version being `removed`.

**procedure** `remove` = (entity *e*, $\text{id}_{\text{version}}$ *v*) boolean *success*:
**begin**
  *success* := FALSE;
  **if** ((`vers-state`(*e*, *v*) = *active*) OR (`vers-state`(*e*, *v*) = *suspended*) OR
          (`vers-state`(*e*, *v*) = *declared*)) **begin**
    `vers-state`(*e*, *v*) := *removed*;
    *success* := TRUE;
  **end**;
**end procedure**
◇

Figure 3.7: Procedure to `remove` an Existing Version from an Entity Derivation Hierarchy

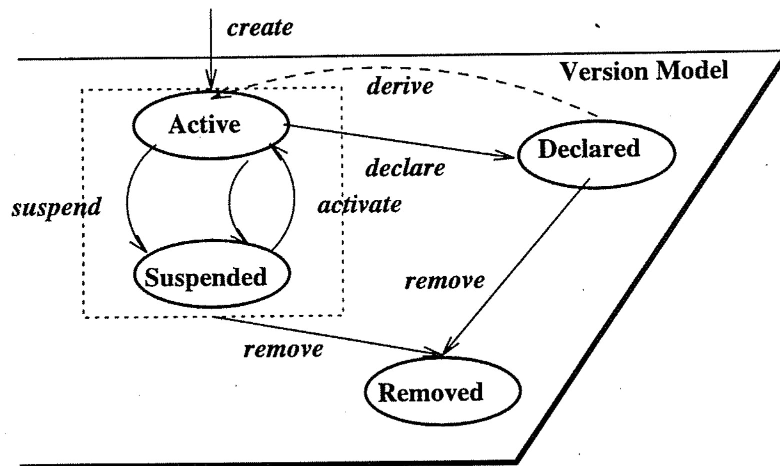



Figure 3.8: Version Model as a Finite State Machine

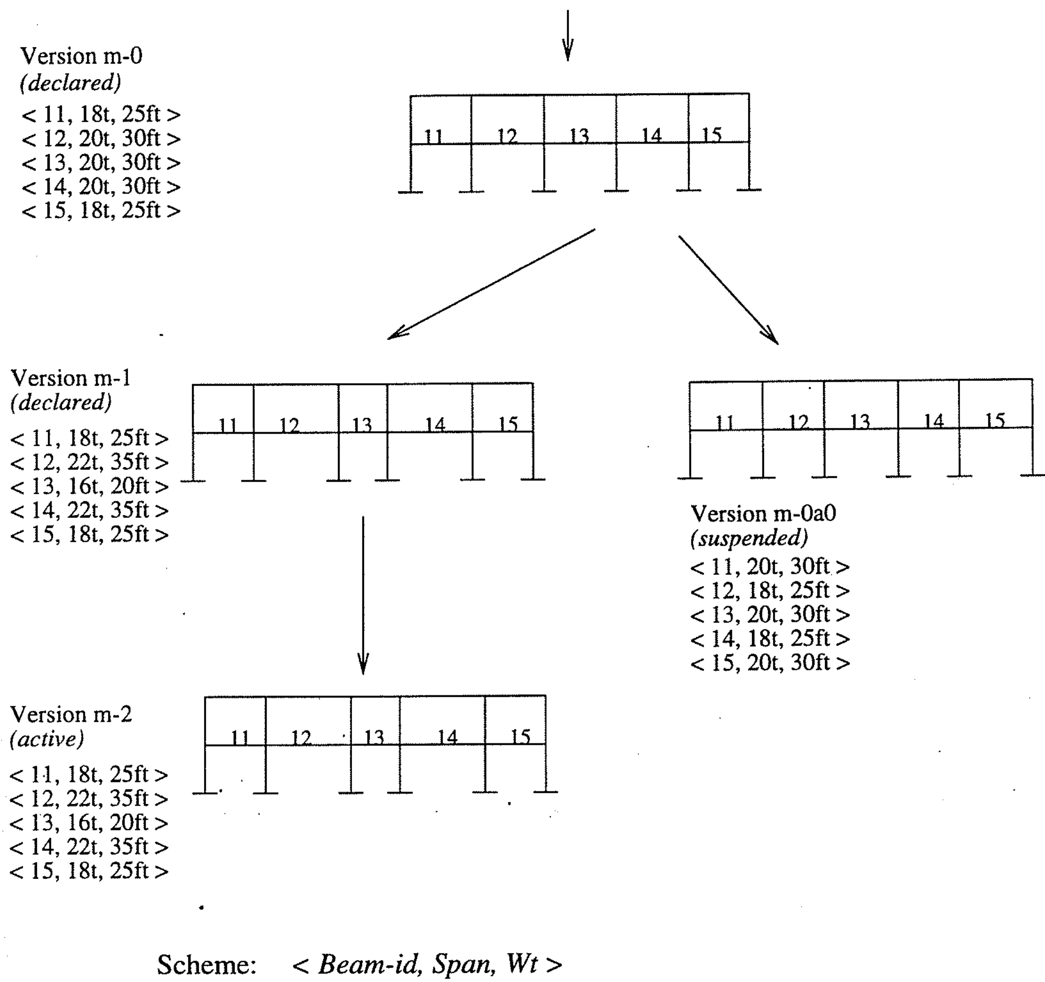



Figure 3.9: BEAM Entity Hierarchy after Executing Operators in Table 3.1

## 3.2 Basic Relational Scheme

We present a scheme to implement the version model in the context of a relational data model. In this paradigm, we describe a primitive entity E by a relation E($\underline{X}$, $Y$), where $X$ denotes the set of *primary* (key) attributes for uniquely identifying a particular instance, and $Y$ denotes the set of *secondary* attributes which describe additional properties of the instance. For clarity, the primary attributes are underlined. Relations presented in this thesis are in BCNF [41], the only non-trivial functional dependencies in a relational scheme are between the primary (key) and the secondary (non-key) attributes.

A scheme to implement the version model must (1) maintain an entity hierarchy, (2) identify the state specified for each version, and (3) describe the contents of versions in the hierarchy. Before describing our versioning scheme, we first present a brief background review of the relational model. Second, we outline the scheme for implementing the version model. Three relations: E-INDEX, E-ACTIVE and E-DATA, completely describe an entity hierarchy. A specific version is uniquely identified by a system attribute, *Version-id*. As mentioned in the previous section, a `vers-numgen` function generates this attribute value when the version was initially `created` or `derived`. We also present SQL implementations of some of the version state operators that were given earlier in Section 3.1. The third part of the chapter proposes a simple protocol, *update-in-place*, to structure interaction between an application and the *active* version. This protocol synchronously executes application changes on the *active* version. We develop a formal framework to reason about changes and outline the procedure to integrate design changes on an entity with the contents of its *active* version. Finally, the fourth part sketches a serious limitation of the current versioning scheme. When `deriving` a new child version, all tuples associated with the parent version are physically copied into the child. Since designers usually modify only a fraction of the tuples while a version is *active*, the scheme stores several copies of the same tuple in two or more versions along the derivation path. This motivates the development of more storage efficient schemes that will be presented in later sections.

### 3.2.1 Overview of the Relational Data Model

In the set-theoretic notion, a *relation* instance is any subset of the Cartesian product of one or more *domains*, where a domain is any (finite) set of values. The members of a relation are called *tuples*. A tuple is an ordered set of values, each one from its respective domain.

Simplistically, a relation can be viewed as a table whose columns are given names called *attributes*, and each row is a tuple. A relation is denoted by a specified schema which is an ordered set of domains and a set of constraints that tuples in the relation must satisfy. *Functional dependency* or key dependency is an example of such a constraint. Finally, a *database schema* is a set of relation schemata, each of which is indexed by relation name. We consider three data operations that manipulate individual tuples: `deletion`, `insertion`, and `replacement`. A `deletion` removes one or more tuples from a relation. On the other hand, an `insertion` is the addition of one or more tuples into a relation. A `replacement` is a combination of deletions and insertions applied in pairs to the same relation performed as a single atomic action that does not require an intermediate consistent state between the deletion and insertion steps.

### 3.2.2 Relational Implementation Scheme

We represent a primitive entity E (described by the scheme E($\underline{X}$, $Y$)) by three relations: E-INDEX, E-ACTIVE, and E-DATA. An E-INDEX relation maintains with each version a link to its parent. As the version set is a tree structure, all versions except the root have exactly one parent; a version tree can be uniquely generated with this information. The E-INDEX relation also identifies versions in the *declared* state. An E-ACTIVE relation locates the currently *active* version. An existing version that is neither *declared* nor *active* is inferred as *suspended*. Finally, an E-DATA relation associates all instances in a version description with its system generated identifier, *Version-id*. The three relations, E-INDEX, E-ACTIVE and E-DATA, are formally defined as follows:

- E-INDEX(*Version-id*, *Parent-id*, *Decl-status*)

  where the attributes specified are given as:

  - *Version-id:* System-generated identifier of the version in an entity derivation hierarchy.
  - *Parent-id:* Identifier of the parent of the given version. As the version set is a tree structure, all versions except the root have exactly one parent.
  - *Decl-status:* Indicator of *declared* versions, which has the possible values: "y" or "n".

- E-ACTIVE(Act-version-id)

  where the attribute *Act-version-id* identifies the currently *active* version. Since an entity can have at most one version in the *active* state, an E-ACTIVE relation has a maximum of one tuple.

- E-DATA(Version-id, X, Y)

  where the attributes specified are given as follows:

  - *Version-id:* System-generated identifier of the version in an entity derivation hierarchy.
  - *X:* Set of primary attributes in the entity scheme that uniquely identify an instance in the given version.
  - *Y:* Set of secondary attributes in the entity scheme that describe certain design properties of each instance. Such attributes in a particular version are functionally dependent on the set of primary attributes.

These three relations completely describe a primitive entity. This representation scheme is more attractive than alternative schemes that identify *active* and *suspended* versions in the E-INDEX relation itself, as in design situations it is typically important to quickly identify the version that is currently *active*. Alternatively, instead of creating an E-ACTIVE table for each entity, a relational table can be constructed by collecting the *active* versions of all entities in that discipline, which requires less storage requirements than maintaining a separate E-ACTIVE relation for each individual entity. Nonetheless, the current implementation employs the E-ACTIVE relation for each entity, mainly for purposes of clarity.

We adopt the following convention for representing relations: Attribute names in a relational scheme are specified in upper case, while specific values assigned to those attributes are in lower case. For example, a BEAM-DATA relation that describes an example BEAM entity has the following definition: BEAM-DATA(Version-id, Beam-id, *Wt*, *Span*) represents the scheme of the relation, while BEAM-DATA(m-0, 11, 18, 25) is a specific tuple with the values m-0, 11, 18, and 25 assigned to the attributes *Version-id, Beam-id, Wt* and *Span*, respectively.

Table 3.2 presents the three relations, BEAM-INDEX, BEAM-ACTIVE, and BEAM-DATA, to maintain the initial version hierarchy of the BEAM entity that was given in Figure 3.1.

In Table 3.2(a), version m-0 is identified as *declared* by the *Decl-status* attribute. Also, the BEAM-ACTIVE relation in Table 3.2(b) specifies version m-0a0 as *active*. In this example, version m-1 is *suspended* as it is neither *declared* nor *active*. Table 3.2(c) shows the BEAM-DATA relation which describes each version in the BEAM derivation hierarchy presented in Figure 3.1.

We use the previously presented example sequence of state operators (Table 3.1) to demonstrate the version model in a relational context. SQL statements for these example operations on the BEAM entity hierarchy are summarized in Table 3.3. The final description of the relations BEAM-INDEX and BEAM-ACTIVE after executing the sequence of operations shown in Table 3.4. The representation of the relations corresponds to the description of the BEAM hierarchy earlier shown in Figure 3.9.

### 3.2.3 Modifying a Version Description

We provide three data operations for modifying instances in a given version: `insert`, `delete` and `replace`. Employing a simple *update in-place* protocol, a single application change on an entity is directly executed on its *active* version. Before describing the procedure to modify an *active* version, we first develop a formal framework to discuss design changes. We then specify criteria to determine the validity of changes in this context. Finally, we develop a procedure, `update-version`, that integrates changes on an entity with the current description of its *active* version, while using the *update-in-place* protocol.

#### Framework for Change Management

In this work, a change is formally defined as a data operation on an instance of an entity. A change on an instance has a *matching tuple* if the instance already exists in the *active* version of the concerned entity. We define the following terms for changes on an entity E, described by the relation E($\underline{X}$, $Y$).

- A *change* A on an instance of entity E (described by the scheme E($\underline{X}$, $Y$)), is maintained as a positive literal described by the scheme, A($\underline{X}$, $Y$, $O$), where
  - $X$: Set of primary attributes that uniquely identify the instance being modified.
  - $Y$: Set of secondary attributes which describe the particular design modification.
  - $O$: Operator descriptor of the change; one of `insert`, `delete`, or `replace`.

- A particular change A, represented as A($\underline{x}$, $y$, $o$), is a data operation $o$, where $o \in$ {`insert`, `delete`, `replace`}, on the instance of primitive entity E that is identified by a set of primary attribute values, $x$.

- A specific change A($\underline{x}$, $y$, $o$) on an instance (identified by primary attribute values, $x$) has a *matching tuple* if the instance already exists in the *active* version of the entity E. This *matching tuple* is specified as: E-DATA(<u>*active-id*</u>, $\underline{x}$, $y'$), where $x$ is the identical set of primary attribute values as the change A. The secondary attribute values of the change, $y$, do not necessarily equal the instance description, $y'$.

- A *sequence* of $n$ changes on an instance of entity E, is represented as a directed graph (chain) G(C,S), where C is the set of changes = $\{C_1, \ldots, C_k, \ldots, C_n\}$, and S is the set of edges between adjacent changes. A specific term in this sequence, $C_k$, corresponds to the $k^{th}$ change in the sequence. We specify that a change $C_i$ is an *earlier* change in the sequence than another change $C_j$, if $i < j$; and the ordering relationship is symbolically denoted as $C_i \prec C_j$. Further, a change $C_i$ *precedes* another change $C_j$, if $i = j$ - 1, and the *precedence* relationship is represented as: $C_i \leadsto C_j$. That is, for $C_i \leadsto C_j$, $C_i$ is an *immediate ancestor* of change $C_j$; alternatively, $C_j$ is an *immediate successor* of change $C_i$.

The above definitions are used throughout the discussions presented in this chapter.

### Validity of data operations

We enumerate rules to specify the validity of a change A on an *active* version of an entity E (described by the relational scheme, E($X$, $Y$)). From a practical viewpoint, a valid change can be successfully integrated with an existing description of an *active* version of the concerned entity E.

- An insert operation A($\underline{x}$, $y$, "insert") is valid only if there <u>does not exist</u> a *matching tuple* in the *active* version. Intuitively, this rule asserts that we can `insert` an instance only if it is not contained in the *active* version.

- A delete operation A($\underline{x}$, $y$, "delete") is valid only if there <u>exists</u> a *matching tuple* in the *active* version. In other words, we can `delete` from an *active* version only those instances which are contained in it.

- A replace operation A(x, *y*, "replace") is valid only if there exists a *matching tuple* in the *active* version. Similar to a `delete` operation, we can `replace` only those instances that already exist in the *active* version.

Further, a sequence of *n* changes represented by a directed graph (chain) G(C,S) is *valid*, if each change in that sequence, $C_i$ where i = 1, ..., *n*, is individually valid.

### Algorithm to integrate changes

We outline a procedure to integrate application changes with the currently *active* version using an *update in-place* protocol. This procedure ensures that each design change on the *active* version is valid based on the rules presented above. We implement this procedure as an `update-version` operator. Table 3.5 shows specific tasks executed for various combinations of a design change A with the possibility of a *matching* tuple in the *active* version, *active-id*, of the E-DATA relation. We illustrate the operator by modifying the *active* version m-2 of our "ongoing" BEAM entity example (previously given in Table 3.4), using the sequences of changes shown in Table 3.6.

1. Sequence 1, Change 1: Matching tuple exists: BEAM-DATA(m-2, 11, 18, 25); `replace` the matching tuple with the new tuple BEAM-DATA(m-2, 11, 20, 30).
2. Sequence 2, Change 1: Matching tuple exists: BEAM-DATA(m-2, 12, 22, 35); `delete` tuple.
3. Sequence 2, Change 2: No matching tuple exists; `insert` tuple BEAM-DATA(m-2, 12, 20, 30).
4. Sequence 3, Change 1: Matching tuple exists: BEAM-DATA(m-2, 14, 22, 35); `replace` the matching tuple with the new tuple BEAM-DATA(m-2, 14, 18, 25).
5. Sequence 3, Change 2: Matching tuple exists: BEAM-DATA(m-2, 14, 18, 25); `replace` the matching tuple with the new tuple BEAM-DATA(m-2, 14, 20, 30).
6. Sequence 4, Change 1: Matching tuple exists: BEAM-DATA(m-2, 15, 18, 25); `replace` the matching tuple with the new tuple BEAM-DATA(m-2, 15, 20, 30).

Table 3.7 describes the BEAM-DATA relation obtained on executing the above data operations.

### 3.2.4 Drawbacks

The proposed scheme suffers from an obvious drawback in that multiple copies of an instance description can occur in many versions along a derivation path. A `derive` operation physically copies all tuples associated with the parent version into the new child version. Since only a fraction of a version's tuples are usually modified while the version is *active*, many copies of an instance can exist in intermediate versions along a derivation path. The BEAM-DATA relation (Table 3.7), for example, stores identical copies of Beam 13 in versions m-1 and m-2. This limitation becomes critical in large derivation hierarchies, having a number of instances (tuples) in each version. In Section 3.4, we present a more compact scheme where a version contains only the summary of all changes made when the version was *active*. The theoretical foundations for this compact scheme are developed in the following section.

Table 3.2: Relational Representation of a BEAM Entity

(a) Maintenance of the BEAM Derivation Hierarchy

BEAM-INDEX

| *Version-id* | *Parent-id* | *Decl-status* |
|---|---|---|
| m-0 | "Null" | "y" |
| m-0a0 | m-0 | "n" |
| m-1 | m-0 | "n" |

(b) Identification of an *Active* Version in the Derivation Hierarchy

BEAM-ACTIVE

| *Act-version-id* |
|---|
| m-0a0 |

(c) Descriptions of Versions in the Derivation Hierarchy

BEAM-DATA

| *Version-id* | *Beam-id* | *Wt* | *Span* |
|---|---|---|---|
| m-0 | 11 | 18 | 25 |
| m-0 | 12 | 20 | 30 |
| m-0 | 13 | 20 | 30 |
| m-0 | 14 | 20 | 30 |
| m-0 | 15 | 18 | 25 |
| m-0a0 | 11 | 20 | 30 |
| m-0a0 | 12 | 18 | 25 |
| m-0a0 | 13 | 20 | 30 |
| m-0a0 | 14 | 18 | 25 |
| m-0a0 | 15 | 20 | 30 |
| m-1 | 11 | 18 | 25 |
| m-1 | 12 | 22 | 35 |
| m-1 | 13 | 16 | 20 |
| m-1 | 14 | 22 | 35 |
| m-1 | 15 | 18 | 25 |

Table 3.3: Database Operations to Specify Version States (Application Example)

| User Operations | SQL Database Operations | |
|---|---|---|
| `activate` (BEAM, m-1) | *active* (initial)<br>*suspended* (initial)<br>operation | m-0a0<br>m-1<br>DELETE FROM BEAM-ACTIVE;<br>INSERT INTO BEAM-ACTIVE<br>VALUES(m-1); |
| `declare` (BEAM, m-1) | *active* (initial)<br>operation | m-1<br>UPDATE BEAM-INDEX<br>SET *Decl-status* = "y"<br>WHERE *Version-id* = m-1; |
| `derive` (BEAM, m-1) | not existent (initial)<br>*declared* (initial)<br>operation | m-2<br>m-1<br>INSERT INTO BEAM-INDEX<br>VALUES(m-2, "n", m-1); |

Table 3.4: Description of the BEAM Entity after Executing the Operations in Table 3.1

(a) Representation of the BEAM Derivation Hierarchy

BEAM-INDEX Relation

| *Version-id* | *Parent-id* | *Decl-status* |
|---|---|---|
| m-0 | "Null" | "y" |
| m-0a0 | m-0 | "n" |
| m-1 | m-0 | "y" |
| m-2 | m-1 | "n" |

(b) Identification of the *Active* Version

BEAM-ACTIVE Relation

| *Act-version-id* |
|---|
| m-2 |

Table 3.5: An `update-version` Operator to Modify a Version for Change A

| Status of physical matching tuple in E-DATA | Change A | | |
|---|---|---|---|
| | A(<u>x</u>, y′, "insert") | A(<u>x</u>, y′, "delete") | A(<u>x</u>, y′, "replace") |
| no matching tuple | `insert` tuple E-DATA(<u>*active-id*</u>, <u>x</u>, y′) | `print` error; | `print` error; |
| E-DATA(<u>*active-id*</u>, <u>x</u>, *y*) | `print` error; | `delete` tuple; | `replace` *matching tuple* with E-DATA(<u>*active-id*</u>, <u>x</u>, y′); |

Table 3.6: Change Made to Instances Contained in an *Active* Version (m-2)

| Sequence-no | Change-no | *Beam-id* | *Wt* | *Span* | *Op-desc* |
|---|---|---|---|---|---|
| 1 | 1 | 11 | 20t | 30ft | "replace" |
| 2 | 1 | 12 | 22t | 35ft | "delete" |
| 2 | 2 | 12 | 20t | 30ft | "insert" |
| 3 | 1 | 14 | 18t | 25ft | "replace" |
| 3 | 2 | 14 | 20t | 30ft | "replace" |
| 4 | 1 | 15 | 20t | 30ft | "replace" |

Table 3.7: Description of the BEAM Entity After Executing the Changes in Table 3.6

BEAM-DATA

| *Version-id* | *Beam-id* | *Wt* | *Span* |
|---|---|---|---|
| m-0 | 11 | 18 | 25 |
| m-0 | 12 | 20 | 30 |
| m-0 | 13 | 20 | 30 |
| m-0 | 14 | 20 | 30 |
| m-0 | 15 | 18 | 25 |
| m-0a0 | 11 | 20 | 30 |
| m-0a0 | 12 | 18 | 25 |
| m-0a0 | 13 | 20 | 30 |
| m-0a0 | 14 | 18 | 25 |
| m-0a0 | 15 | 20 | 30 |
| m-1 | 11 | 18 | 25 |
| m-1 | 12 | 22 | 35 |
| m-1 | 13 | 16 | 20 |
| m-1 | 14 | 22 | 35 |
| m-1 | 15 | 18 | 25 |
| m-2 | 11 | 20 | 30 |
| m-2 | 12 | 20 | 30 |
| m-2 | 13 | 16 | 20 |
| m-2 | 14 | 20 | 30 |
| m-2 | 15 | 20 | 30 |

## 3.3 Theory of Equivalent Operations

An *equivalent operation* for a sequence of changes on an instance is a single data operation that results in the same final description of the instance as the original sequence of changes. This section establishes that we can compute a unique *equivalent operation* for a valid sequence of changes, from knowing the first and last changes in that sequence. Before we can prove this result, we need to develop a framework in which to reason about *equivalent operations.* We first present the basic rules to compute an *equivalent operation* for a pair of changes on an instance of entity E. We then extend this framework for an arbitrarily long sequence of changes by establishing that for any valid sequence of changes on an instance, there exists at most one *equivalent operation* which is also valid. As a corollary to this result, the last change in a valid sequence of changes describes the instance that has been modified by that sequence. Using the above results, we develop important algebraic laws that establish equivalences between valid operator sequences. Finally, we show that the *equivalent operation* for a sequence of operators can be uniquely determined by using only its first and last elements. The framework for *equivalent operations* developed in this section provides the theoretical foundation for version change management.

### 3.3.1 Rules to Compute an Equivalent Operation

We enumerate the following rules to compute an *equivalent operation* Q (described by the scheme $Q(\underline{X}, Y, O)$) for an ordered pair of two changes $A(\underline{x}, y, o)$ and $B(\underline{x}, y', o')$ on an instance of entity E that is identified by the primary attribute values, $x$. For this pair of changes, A is an *immediate ancestor* of the change B, i.e., $A \leadsto B$. Note that the descriptions of the two changes (secondary attribute values, $y$ and $y'$) are typically different unless otherwise specified. Further, we assume A to be a valid change; the validity of a given pair of changes is thus determined by the individual validity of change B in the context of change A.

1. $A(\underline{x}, y,$ "insert"$) \leadsto B(\underline{x}, y',$ "insert"$)$ is an invalid sequence. B is an invalid operation since we cannot `insert` an instance that already exists; the instance has been `inserted` by operation A.

2. $A(\underline{x}, y,$ "insert"$) \leadsto B(\underline{x}, y',$ "delete"$) \Leftrightarrow$ no operation. Note that in this case $y$ is the same set of values as $y'$.

3. A($\underline{x}$, $y$, "insert") $\rightsquigarrow$ B($\underline{x}$, $y'$, "replace") $\Leftrightarrow$ Q($\underline{x}$, $y'$, "insert")

4. A($\underline{x}$, $y$, "delete") $\rightsquigarrow$ B($\underline{x}$, $y'$, "insert") $\Leftrightarrow$ Q($\underline{x}$, $y'$, "replace")

5. A($\underline{x}$, $y$, "delete") $\rightsquigarrow$ B($\underline{x}$, $y'$, "delete") is an invalid sequence. B is an invalid operation since we cannot `delete` an instance that no longer exists; the instance has been `deleted` by operation A.

6. A($\underline{x}$, $y$, "delete") $\rightsquigarrow$ B($\underline{x}$, $y'$, "replace") is an invalid sequence. B is an invalid operation since we cannot `replace` an instance that no longer exists; the instance has been `deleted` by operation A.

7. A($\underline{x}$, $y$, "replace") $\rightsquigarrow$ B($\underline{x}$, $y'$, "insert") is an invalid sequence. B is an invalid operation since we cannot `insert` an instance that already exists; A was a valid operation because the instance already existed.

8. A($\underline{x}$, $y$, "replace") $\rightsquigarrow$ B($\underline{x}$, $y'$, "delete") $\Leftrightarrow$ Q($\underline{x}$, $y'$, "delete"). Note that in this case $y$ is the same set of values as $y'$.

9. A($\underline{x}$, $y$, "replace") $\rightsquigarrow$ B($\underline{x}$, $y'$, "replace") $\Leftrightarrow$ Q($\underline{x}$, $y'$, "replace")

The equivalence rules show that for a valid sequence of two operations (the sequence is valid if both changes A and B are valid) that do not result in a "no operation", there exists a single *equivalent operation*. It can be shown that such an *equivalent operation*, Q is also valid. The equivalent rules can be implemented as a function, `get-eqchange`, that computes the *equivalent operation*, if any, for a valid sequence of two operations.

### 3.3.2 Equivalent operation of a sequence of operations

We extend *equivalent operations* for an arbitrarily long valid sequence of operations, by guaranteeing that we can summarize the sequence of operations on an instance into at most one *equivalent operation.*

**Theorem 1** *For any valid sequence of data operations on an instance, there exists at most one equivalent operation* $Q_{seq}$ *which is also valid. This equivalent operation produces the same final description of the instance as the execution of the original sequence of the data operations.*

**Proof:** We represent a sequence of $n$ changes on an instance of entity E (identified by a set of primary attribute values, x) as a directed chain G(C,S), where C = $\{C_1, \ldots, C_k, \ldots C_n\}$ is the set of changes, and S is the set of edges between adjacent changes. A specific node in this sequence, $C_k$, denotes the $k^{th}$ term in the sequence and is described as $C_k(\underline{x}, y, o)$. A directed edge $S_i$ in S represents the transit from an earlier change $C_i$ in the sequence to its immediate successor change in the sequence $C_{i+1}$ ($C_i \rightsquigarrow C_{i+1}$).
*Base:* If there exists only one operation in the sequence G(C,S), the *equivalent operation* is that operation itself.

*Proof Steps:* For an arbitrarily long sequence G (of length more than one), we show that the sequence can be recursively compressed to a single node which is the *equivalent operation.* The rest of this proof outlines the steps to compress the sequence, as well as the rationale for the resulting *equivalent operation* to be valid.

1. Start with the root node $C_0$ of the chain G.

2. If there exists a node $C_1$ adjacent to the root node $C_0$ ($C_0 \rightsquigarrow C_1$) then use the rules specified in Section 3.3.1 to compute the equivalent change $Q_0$ for the sequence $C_0 \rightsquigarrow C_1$. Substitute in the chain G, the sequence of two operations $C_0 \rightsquigarrow C_1$ by the equivalent operation $Q_0$. Rename $Q_0$ as the new root node $C_0$. By the equivalence rules, if both $C_0$ and $C_1$ are valid, then there exists at most one equivalent operation that is also valid (the equivalent operation could be a "no operation"). Furthermore, as the equivalent node has the same effect as the two operations that it substituted, the successor node of $C_1$ that was initially valid would continue to remain valid.

3. Repeat Step 2 till there does not exist any node adjacent to the root node, i.e, the chain has been collapsed to a single node. As the sequence is valid, each execution of the `get-eqchange` operator is guaranteed to obtain a valid equivalent operation. The resulting root node $C_0$ is the operation $Q_{seq}$ that is equivalent to the sequence of operations.

◇
The result of this theorem establishes that there exists a unique *equivalent operation* for a valid sequence of changes.

### 3.3.3 Final Description of Instance

A corollary of Theorem 1 ensures that the computed *equivalent operation* for a valid sequence of operations is described by the last change.

**Corollary of Theorem 1:** *The description of an instance, after the execution of a valid sequence of data operations on it, is determined by the last change in that sequence.*
◇
Although we do not present a formal proof for this result, we intuitively explain it using the following argument. The last change of the concerned valid sequence could be one of: `insert`, `delete`, or `replace` operations. For each case we can show that the final description of the instance is obtained by the last change.

- `delete`: If the last change is a `delete` operation, then the *equivalent operation* results in an non-existent description of the instance.
- `insert` or `replace`: If the last change is either an `insert` or `replace` operation, then the final description of the instance corresponds to the inserted or final replaced values, both of which are specified by the last change.

Theorem 1 and its Corollary establish that for a valid sequence of changes, there exists at most one *equivalent operation* which is also valid. Furthermore, the resulting description of the instance is determined by the last change in the original sequence. This result provides the theoretical basis for the methodologies presented later in this chapter for version change management.

### 3.3.4 Algebraic Laws for Precedence Relationships

We now specify important algebraic laws for equivalences among valid operator sequences connected pairwise by precedence relationships. Let's denote three valid data operations, A, B, and C, on an individual instance of an entity E. We provide a connector, $\Leftrightarrow$, to signify equivalences between the operator expressions that it connects. Using the standard notations that were introduced in Section 3.2.3, we show the following four algebraic laws among the changes A, B and C.

1. *Identity Law:* (A $\Leftrightarrow$ A $\leadsto$ "no operation" $\Leftrightarrow$ "no operation" $\leadsto$ A) If A is a valid operation then it is also valid if it is either preceded or succeeded by a "no operation."

Further, if A is an invalid change, then the other two operator sequences (containing A and "no operation") are also invalid.

2. *Idempotent Law* does not hold: ($A \rightsquigarrow A \not\Leftrightarrow A$) Even if A is a valid operation, $A \rightsquigarrow A$ may not be valid. It can be easily shown that $A \rightsquigarrow A$ is valid only when A is a `replace` operation.

3. *Commutative Law* does not hold: ($A \rightsquigarrow B \not\Leftrightarrow B \rightsquigarrow A$) If $A \rightsquigarrow B$ is a valid sequence, it does not imply that $B \rightsquigarrow A$ is also valid. Moreover, even if both expressions are valid, then by Corollary 1, operation B determines the instance description after the execution of the left expression, while in the right expression, operation A controls the final instance description. This implies that unless A and B are identical `replace` operations (considering only situations where $A \rightsquigarrow B$ and $B \rightsquigarrow A$ are both valid), their resulting descriptions are likely to be different.

4. *Associative Law:* ($(A \rightsquigarrow B) \rightsquigarrow C \Leftrightarrow A \rightsquigarrow (B \rightsquigarrow C)$) If the sequence $A \rightsquigarrow B \rightsquigarrow C$ is valid, then from the result of Theorem 1, the sequence results in at most one equivalent operation. It can be shown that getting this unique equivalent operation is independent of the order in which the equivalences are computed.

These laws are used in developing valid algorithms for operators to detect changes in an application environment and to integrate application changes on an entity with the current description of its *active* version.

### 3.3.5 Computing Equivalent Operation for a Valid Sequence of Changes

We are now ready to show that it is possible to uniquely determine the *equivalent operation* for any arbitrary sequence of operators, from the first and last elements of that sequence. Before stating this result, we prove the following lemma.

**Lemma 1** *: For a valid sequence of operations on an instance, given the first and the last operations in that sequence, there exists at most one possible equivalent operation* $Q_{int}$ *("no operation" (nop) is also a possible outcome) for the intermediate operations in the sequence. This equivalent operation is called the equivalent intermediate operation.*

**Proof:** We represent a sequence of $n$ changes on an instance of entity E (identified by a set of primary attribute values, $x$) as a directed chain G(C,S), where $C = \{C_1, \ldots, C_k, \ldots C_n\}$

is the set of changes, and S is the set of edges between adjacent changes. A specific node in this sequence, $C_k$, denotes the $k^{th}$ term in the sequence and is described by the scheme, $C_K(\underline{x}, y, o)$. A directed edge $S_i$ in S represents the transit from an earlier change $C_i$ in the sequence to its immediate successor change in the sequence $C_{i+1}$ ($C_i \rightsquigarrow C_{i+1}$).

The reasoning behind this proof is as follows. For a valid sequence of $n$ operations ($C_1, \ldots, C_n$) in the sequence G, we can, as a result of Theorem 1, summarize all the intermediate changes ($C_2, \ldots, C_{n-1}$) into a single unique equivalent intermediate operation $Q_{int}$. The proof for this lemma, thus, translates into showing that for specific assignments of the first ($C_1(\underline{x}, y,$ *op-desc*)) and last ($C_n(\underline{x}, y',$ *op–desc'*)) operators in the valid sequence, there exists at most one definition for the equivalent operation $Q_{int}$.

*Proof Steps:* The proof presented here is exhaustive in that it analyzes the different possible operator assignments for changes $C_1$ and $C_n$ and uses the equivalence rules to show that for each of the possible combination of assignments for the first and last changes ($C_1$ and $C_n$), there is at most one feasible operator assignment for the equivalent operation $Q_{int}$ for the entire sequence to be valid. The actual steps for proving each case is as follows: (i) We first ignore the operator assignment for the first change $C_1$ and determine all possible operator assignments for $Q_{int}$, such that the operator assignment of the change $C_n$ remains valid. (ii) We then select from this list of possible operator assignments for $Q_{int}$, a smaller subset of operator assignments which would also ensure that change $C_1$ is a valid operation.

- $C_1(\underline{x}, y,$ "insert") and $C_n(\underline{x}, y',$ "insert"): For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y,$ "delete") or "no operation" (nop). Since $C_1$ and $C_n$ are both valid `insert` operations, the equivalence rules in Section 3.3.1 eliminate the possibility that the intermediate operations result in a "no operation." $Q_{int}(\underline{x}, y,$ "delete") is the only allowable intermediate equivalent operation for the entire sequence to be valid.

- $C_1(\underline{x}, y,$ "insert") and $C_n(\underline{x}, y',$ "delete"): For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y',$ "insert") or $Q_{int}(\underline{x}, y',$ "replace"). Since $C_1$ is an `insert` operation and the intermediate equivalent operation $Q_{int}$ is valid, the equivalence rules in Section 3.3.1 eliminate the possibility that $Q_{int}$ is also an `insert` operation. $Q_{int}(\underline{x}, y',$ "replace") is thus the only allowable intermediate equivalent operation for the entire sequence to be valid.

- $C_1(\underline{x}, y,$ "insert") and $C_n(\underline{x}, y',$ "replace"): For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y'',$ "insert") or $Q_{int}(\underline{x}, y'',$ "replace"). Since $C_1$ is an `insert` operation and the intermediate equivalent operation $Q_{int}$ is valid, the equivalence rules in Section 3.3.1 eliminate the possibility that $Q_{int}$ is also an `insert` operation. $Q_{int}(\underline{x}, y',$ "replace") is thus the only allowable intermediate equivalent operation for the entire sequence to be valid.

- $C_1(\underline{x}, y,$ "delete") and $C_n(\underline{x}, y',$ "insert"): For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y,$ "delete") or "no operation" (nop). Since $C_1$ is a `delete` operation and the intermediate equivalent operation $Q_{int}$ is valid, the equivalence rules in Section 3.3.1 eliminate the possibility that $Q_{int}$ is also a `delete` operation. Thus, the equivalent intermediate operation must result in a "no operation" (nop) for the entire sequence to be valid.

- $C_1(\underline{x}, y,$ "delete") and $C_n(\underline{x}, y',$ "delete"): For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y',$ "insert") or $Q_{int}(\underline{x}, y',$ "replace"). Since $C_1$ is a `delete` operation and the intermediate equivalent operation $Q_{int}$ is valid, the equivalence rules in Section 3.3.1 eliminate the possibility that $Q_{int}$ is a `replace` operation. $Q_{int}(\underline{x}, y',$ "insert") is thus the only allowable intermediate equivalent operation for the entire sequence to be valid.

- $C_1(\underline{x}, y,$ "delete") and $C_n(\underline{x}, y',$ "replace"): For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y'',$ "insert") or $Q_{int}(\underline{x}, y'',$ "replace"). Since $C_1$ is a `delete` and the intermediate equivalent operation $Q_{int}$ is valid, the equivalence rules in Section 3.3.1 eliminate the possibility that $Q_{int}$ is a `replace` operation. $Q_{int}(\underline{x}, y',$ "insert") is thus the only allowable intermediate equivalent operation for the entire sequence to be valid.

- $C_1(\underline{x}, y,$ "replace") and $C_n(\underline{x}, y',$ "insert"): For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y,$ "delete") or "no operation" (nop). Since $C_1$ is a `replace` operation and both the intermediate equivalent operation $Q_{int}$ and the last change $C_n$ are both valid, the equivalence rules in Section 3.3.1 eliminate the possibility that $Q_{int}$ results in a "no operation." $Q_{int}(\underline{x}, y',$ "delete") is thus the only allowable intermediate equivalent operation for the entire sequence to be valid.

- $C_1(\underline{x}, y, \text{"replace"})$ and $C_n(\underline{x}, y', \text{"delete"})$: For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y', \text{"insert"})$ or $Q_{int}(\underline{x}, y', \text{"replace"})$. Since $C_1$ is a `replace` operation and the intermediate equivalent operation $Q_{int}$ is valid, the equivalence rules in Section 3.3.1 eliminate the possibility that $Q_{int}$ is an `insert` operation. $Q_{int}(\underline{x}, y', \text{"replace"})$ is thus the only allowable intermediate equivalent operation for the entire sequence to be valid.

- $C_1(\underline{x}, y, \text{"replace"})$ and $C_n(\underline{x}, y', \text{"replace"})$: For $C_n$ to be valid, the intermediate equivalent operation must be either $Q_{int}(\underline{x}, y'', \text{"insert"})$ or $Q_{int}(\underline{x}, y'', \text{"replace"})$. Since $C_1$ is a `replace` operation and the intermediate equivalent operation $Q_{int}$ is valid, the equivalence rules in Section 3.3.1 eliminate the possibility that $Q_{int}$ is an `insert` operation. $Q_{int}(\underline{x}, y', \text{"replace"})$ is thus the only allowable intermediate equivalent operation for the entire sequence to be valid.

$\diamond$

With Lemma 1, we can now state the main result in the following theorem.

**Theorem 2** : *For any valid sequence of operations on an instance, given the first and last operators, it is possible to determine the value of the equivalent operation* $Q_{seq}$ *for the sequence.*

**Proof:** We represent a sequence of $n$ changes on an instance of entity E (identified by a set of primary attribute values, $x$) as a directed chain G(C,S), where $C = \{C_1, \ldots, C_k, \ldots C_n\}$ is the set of changes, and S is the set of edges between adjacent changes. A specific node in this sequence, $C_k$, denotes the $k^{th}$ term in the sequence and is described as $C_k(\underline{x}, y, o)$. A directed edge $S_i$ in S represents the transit from an earlier change $C_i$ in the sequence to its immediate successor change in the sequence $C_{i+1}$ ($C_i \rightsquigarrow C_{i+1}$).
*Base:* If there exists exactly one operation in the sequence G, then the *equivalent operation* for the sequence $Q_{seq}$ is that operation itself. If, however, the sequence G has two operations, we can uniquely compute the unique *equivalent operation* using the rules given in Section 3.3.1 (summarized as a `get-eqchange` function).

*Proof Steps:* The proof for an arbitrarily long sequence G of $n$ (three or more) operators has the following steps:

1. Given the results of Lemma 1 and the first ($C_1$) and last ($C_n$) changes in the operator sequence, G, we can summarize all the intermediate changes ($C_2$,...,$C_{n-1}$) into one valid equivalent intermediate operation $Q_{int}$. In other words, we reduce the original sequence of $n$ operations (G) into the following sequence of 3 operations: ($C_1$, $Q_{int}$, $C_n$).

2. For each combination of the above sequence of three valid changes ($C_1$, $Q_{int}$, $C_n$) we can determine the net equivalent operation, $Q_{seq}$, by using two times the equivalence rules given in Section 3.3.1. Theorem 1 guarantees that $Q_{seq}$ is both unique and valid. Furthermore, by the Corollary of Theorem 1, the final description of the equivalent operation, $Q_{seq}$, is obtained by the last change ($C_n$).

◇

This result provides the theoretical basis for procedures to compute the net differences between two versions, one of which is an *ancestor* of the other in an entity derivation hierarchy. We compute the net differences as a set of *equivalent operations* that can be executed on the *ancestor* version to describe its *descendant*. Using Theorem 2, we can obtain the *equivalent operation* on each instance from only the first and last change on the instance along the derivation path. In other words, such a procedure does not require the intermediate descriptions of each modified instance, making it particularly attractive for complex design situations that typically maintain a large number of intermediate descriptions of an incrementally refined artifact.

## 3.4 Representation Scheme Based on Storing Changes

This section proposes a compact *forward deltas* scheme [36] to implement the version hierarchy of a primitive entity. Each version in this scheme contains a summary of all changes made to instances of the entity when the concerned version was *active*. Thus, a given version can be described by executing its associated changes (set of *deltas*) on the description of its parent version. By applying the concept of *equivalent operations*, we establish a version description as the set of *equivalent operations* of all changes on that entity that were made when the given version was *active*. A version, in our model, is thus a unit of granularity whose consistency can be evaluated. For this scheme, `deriving` a new version does not produce a physical copy of the parent version's tuples in the child version. Instead, the child version logically *inherits* the description of the parent at the time it is `derived`.

In this section, we first extend the basic representation scheme presented in Section 3.2.2, to implement versions as a set of *equivalent operations*. We then discuss a procedure to describe versions that are represented by the proposed scheme. Tuples logically belonging to a version description could be either associated with the particular version or *inherited* from its *ancestors*. However, the proposed procedure is computationally inefficient since in order to instantiate a version, the operator retraces back to the root version of the derivation hierarchy. A storage for efficiency tradeoff includes explicitly storing instantiated descriptions of intermediate versions; we denote the instantiated versions as *complete*, the remaining versions are *incomplete*. The modified procedure to instantiate a version now needs to trace only till the most recent *complete ancestor* version which contains all tuples from its own *ancestors* that could be potentially *inherited* by the version being described.

### 3.4.1 Basic Representation Scheme

For each version, we associate its *Version-id* attribute value with a set of *equivalent operations*, at most one on each instance that was modified when that version was *active*. To completely describe the *equivalent operations* contained in a given version, we extend the definition of the E-DATA relation presented in Section 3.2.2 by adding two new attributes, *Op-desc* and *Vprev*. The *Op-desc* attribute records the descriptor of the *equivalent operation* on each instance. On the other hand, an *Vprev* attribute maintains with an instance a link to its most recent description in an *ancestor* version. The *Vprev* attribute value is critical in redesign efforts that require retracing of design changes through *ancestor* versions

```
SQL> select * from Beam_Data;

VERSION_ID   BEAM_ID          WT         SPAN OP_DESC    VPREV
----------  ----------  ----------  ---------- ---------- ----------
m-0                 11          18          25 insert     Null
m-0                 12          20          30 insert     Null
m-0                 13          20          30 insert     Null
m-0                 14          20          30 insert     Null
m-0                 15          18          25 insert     Null
m-0a0               11          20          30 replace    m-0
m-0a0               12          18          25 replace    m-0
m-0a0               14          18          25 replace    m-0
m-0a0               15          20          30 replace    m-0
m-1                 12          22          35 replace    m-0
m-1                 13          16          20 replace    m-0

VERSION_ID   BEAM_ID          WT         SPAN OP_DESC    VPREV
----------  ----------  ----------  ---------- ---------- ----------
m-1                 14          22          35 replace    m-0

12 rows selected.

SQL> 
```

Figure 3.10: Extended Representation Scheme of the BEAM Hierarchy Shown in Figure 3.9

till when the unsatisfactory change was first introduced. For the sake of performance, we explicitly store this value instead of computing the value every time it is needed. The modified definition of an E-DATA relation is now given:

E-DATA(<u>*Version-id*</u>, <u>*X*</u>, *Y*, *Op-desc*, *Vprev*)

where the additional attributes are:

- *Op-desc:* Descriptor of the *equivalent operation* of all changes on a particular instance when the given version was *active*. Possible values are: "insert," "delete," or "replace."
- *Vprev:* Identifier of the ancestor version in the derivation hierarchy in which the instance was most recently modified. If the instance was `inserted` in the given version, then the *Vprev* attribute is assigned a value, "Null."

Figure 3.10 uses the current scheme to implement the BEAM-DATA relation in an ORACLE database. This relation describes the BEAM hierarchy that was described earlier in Figure 3.9 (Section 3.1).

### 3.4.2 Describing a Version Definition

We outline a procedure that describes versions which are represented by the proposed *forward deltas* scheme. This procedure collects all tuples that logically belong to a version description; a particular tuple could be either associated with the current version or *inherited* from its ancestors.

Previous research efforts have proposed a number of algorithms to instantiate versions represented using similar *forward deltas* schemes [36]. These algorithms usually start with the root version and successively apply the *deltas* associated with each intermediate version on the derivation path from the root version to (and including) the version being instantiated. However, such algorithms implicitly require identifying the actual derivation path from the root version to the concerned version. Unfortunately, search procedures that identify derivation paths could themselves be computationally significant in large design hierarchies with a high degree of branching.

Given that a version (except the root version) in an entity hierarchy has exactly one parent, we propose an alternate procedure that retraces the parent links from (and including) the concerned version to the root version collecting tuples that logically belong to the instantiated version. We use Corollary 1 (Section 3.3.3) for identifying specific tuples that logically belong to the version of interest. A version derivation path can be viewed as a sequence of *equivalent operations* on each instance that was modified when the concerned version or one of its *ancestors* were *active*. The description of each instance that is *included* in a given version definition is therefore determined by the last element in its derivation sequence. If this is a `delete` operation, then that instance is not present in the version description. However, all the latest non-delete *equivalent operations* are included while describing the version of interest. Further, included tuples that are physically associated with a *proper ancestor* of the given version are denoted as *inherited*. We implement the above procedure as a `materialize` operator that describes a version by obtaining the latest *equivalent operation* on each instance along the version's derivation path. Figure 3.11 outlines an algorithm for the `materialize` operator. In addition to the type declarations in Section 3.1, we use the following type declarations:

- tuple: Relational description of a change on an instance of an entity E that is stored in an E-DATA relation.
- secondary: Set of secondary attributes that describe the design properties of each instance of an entity E.

- opdesc: Descriptor of the *equivalent operation* on an instance stored in a particular version of an entity E.

In addition, this algorithm and subsequent algorithms use the following *external functions*:

- `vers-type` = (entity E, $\text{id}_{\text{version}}$ V) string S: Specifies whether a given version V of entity E is *complete*. Possible values for the string S are *complete* or *incomplete*.
- `fetch` = (entity E, $\text{id}_{\text{version}}$ V, primary K) tuple T: Queries a version V in an E-DATA relation for an instance that is identified by the primary attributes K. The queried result is a tuple T from the E-DATA relation.
- `retrieve` = (entity E, $\text{id}_{\text{version}}$ V) tuple T: Obtains each tuple, in sequence, that is physically associated with version identifier V in relation E-DATA. An execution of this function returns a single tuple T that represents the instance which is subsequent to the instance that was retrieved by the most recent execution of the `retrieve` function for the same version V.
- `key` = (tuple T) primary K: Returns the primary attribute values K of a tuple T.
- `dependent` = (tuple T) secondary Y: Returns the secondary attribute values Y of a tuple T.
- `operator` = (tuple T) opdesc O: Returns the operator descriptor O of a tuple T.
- `output`: Prints the assigned parameter values.
- $\pi_F$T: Corresponds to the relational algebra operator, **project**. It obtains values assigned to attributes F in tuple T.
- `add` = (primary X, [1:n] primary H) primary H: Adds an primary attribute X into the appropriate position of a sorted array of primary attributes H. The resulting sorted array replaces the original array H.

Table 3.8 illustrates a `materialize` operation by describing *active* version m-2 of our example BEAM version hierarchy that was shown in Figure 3.10.

### 3.4.3 Optimization for Representation Scheme

The implementation scheme for versions as a set of *equivalent operations* is efficient in terms of storage. However, each time we instantiate a version, a `materialize` operation must retrace back to the root version of the derivation hierarchy. Our approach to improve the computational efficiency is to explicitly store instantiated descriptions of certain intermediate versions in the E-DATA relation. The "optimized" scheme denotes the instantiated

Table 3.8: Instantiation of Version m-2: `materialize`(m-2)

| *Beam-id* | *Wt* | *Span* |
|---|---|---|
| 14 | 22 | 35 |
| 13 | 16 | 20 |
| 12 | 22 | 35 |
| 15 | 18 | 25 |
| 11 | 18 | 25 |

versions as *complete*, while the remaining versions are *incomplete*. By this storage for efficiency tradeoff, a `materialize` operation to instantiate an *incomplete* version needs to trace only until the most recent *complete ancestor* version. As this *ancestor* version is *complete*, it contains all tuples from its own *ancestors* that could be potentially *inherited* by the version being instantiated. Further, a *complete* version can be described by simply querying the E-DATA relation on its version identifier.

To represent this "optimized" scheme, we need to distinguish *complete* versions in an entity hierarchy, as well as identify those tuples in a *complete* version that were explicitly copied from its *ancestor* versions. We accomplish these two requirements by the following extensions to the representation scheme proposed in Section 3.4.1.

- A new value, "copy," is added to the set of possible values that can be assigned to attribute, *Op-desc*, in the E-DATA relation. This attribute value identifies those tuples that are explicitly copied into a *complete* version from one of its *ancestors*.
- A new attribute, *Version-type*, in the scheme for the E-INDEX relation identifies versions that are *complete*. The root version of an entity derivation hierarchy is always specified as *complete*.

An *incomplete* version can be made *complete* by explicitly copying the *inherited* tuples that logically belong to the version description into its definition. The *Op-desc* attribute of each copied tuple is assigned a value, "copy." Also, the *Version-type* attribute for the given version in the E-INDEX relation is assigned a value, "complete." Figure 3.12 describes, in detail, a procedure to `complete` a version. The next section discusses the application of *equivalent operations* for version change management.

**Algorithm 7** (`materialize`)

*Input:* $e$: Name of the concerned entity.

*materialize-id*: identifier of the version being `materialized`.

*Output:* All the tuples that logically belong to version identified by *materialize-id*

```
procedure materialize = (entity e, id_version materialize-id):
begin tuple u; primary x; secondary y; [1:n] primary h; id_version version-id; integer i;;
  for i = 0 by 1 to n begin
    h[i] := 0;
  end;
  version-id := materialize-id;
  while (version-id ≠ "Null") do
  begin
    u := retrieve(e, version-id);
    while (u ≠ "Null") do
    begin
      if ((x := key(u)) ∉ h)
      begin
        add(x, h);
        if (operator(u) ≠ "delete")
        begin
          output(π_{x,y} u);
        end;
      end;
      u := retrieve(e, version-id);
    end;
    version-id := vers-parent(version-id);
  end;
end procedure
◇
```

Figure 3.11: Algorithm to `materialize` a Version Represented by Equivalent Operations

**Algorithm 8** (complete)

*Input:* *e:* Name of the concerned entity.
*complete-id:* Identifier of the version being completed.

*Output:* Relation E-DATA($\underline{X}$, $\underline{Y}$, *O*, *A*) with version *complete-id* instantiated.

```
procedure instantiate-tuple = (id_version version-id, tuple u) boolean success:
begin primary x; [1:n] primary h;;
   if ((x := key(u)) ∉ h) begin
      add(x, h);
      if (operator(u) ≠ "delete" AND version-id ≠ complete-id) begin
         success:= TRUE;
      end;
   end;
end procedure

procedure complete = (entity e, id_version complete-id):
begin id_version version-id, tuple u, primary x, secondary y;
   version-id := complete-id;
   while (version-id ≠ "Null" AND vers-type(version-id) ≠ "complete") do
   begin
      u:= retrieve(version-id);
      while (u ≠ "Null") do
      begin
         success:= instantiate-tuple(version-id, u);
         x:= key(u);
         y:= dependent(u);
         if (success) begin
            insert E-DATA(complete-id, x, y, version-id);
         end;
         u:= retrieve(version-id);
      end;
      version-id := vers-parent(version-id);
   end;
   u:= retrieve(version-id);
   while (u ≠ "Null") do
   begin
      success:= instantiate-tuple(version-id, u);
      x:= key(u);
      y:= dependent(u);
      if (success) begin
         insert E-DATA(complete-id, x, y, version-id);
      end;
      u:= retrieve(version-id);
   end;
end procedure
◇
```

Figure 3.12: Procedure to complete a Version in an Entity Derivation Hierarchy

## 3.5 Application of *Equivalent Operations* to the Version Model

We apply *equivalent operations* to store, detect, and manage changes among versions in an entity derivation hierarchy. As developed in the previous section, a given version contains the set of *equivalent operations* on the instances which were modified when that version was *active*. Therefore, to modify an *active* version we (i) determine the net changes made to an entity during an application session, and (ii) integrate these detected changes with the current description of the *active* version. The exact methodology to detect net changes depends on the specific protocol being employed to structure the interaction between the application session and the version hierarchy.

We introduce a *check-out/check-back-deltas* protocol that structures the interaction between an application and the version hierarchy. Using a *check-out/check-back-deltas* protocol, a designer *checks-out* a `materialized` description of the *active* version into the application environment. This description can then be iteratively refined using three primitive operations: `insert`, `delete`, and `replace`. A `compress` operation summarizes the changes made on instances of the entity during that application session. The net change on a particular instance being the *equivalent operation* on that instance that was made during the concerned session. Finally, we *check-back* these detected changes on an entity into its *active* version, where they are `integrated` with the present version description. Since an *active* version could be modified by several application sessions, a given version contains the *equivalent operations* of all changes that were made to that entity while that version was *active*. Our model therefore specifies a version as a unit of granularity. In addition, we can compute the changes between an *ancestor-descendant* pair of versions in an entity hierarchy by determining the net *equivalent operation* of all intermediate versions along the path between the two concerned versions. The resulting changes represent the minimal set of data operations that can be executed on the concerned *ancestor* version to describe its *descendant*.

The rest of this section discusses, in detail, specific procedures to support the design process for a primitive entity. First, we present a procedure to `materialize` a version that is represented using the "optimized" scheme presented in Section 3.4.3. Second, we outline the procedure to detect the *equivalent operations* made during a particular application session using a *check-out/check-back-deltas* protocol. Third, we discuss a procedure to `integrate` detected application changes on each entity with the existing description of its *active* version.

The fourth part of this section shows a procedure to `compute` the differences between an *ancestor-descendant* pair of versions in the entity derivation hierarchy. As mentioned earlier, we need to prune a version hierarchy for curtailing its storage needs. In Section 3.1, we outlined a general procedure to reconfigure a version hierarchy when `removing` versions from it. The last part of this section focuses on procedures to erase the contents of a *removed* version and also sketches an SQL implementation for `removing` a version definition from an entity hierarchy. We demonstrate the proposed change management operators by using our "ongoing" example of a BEAM entity derivation hierarchy.

### 3.5.1 Describing a Version

A `materialize` operation creates a version representation by collecting all instances that logically belong to it; the instances could be either modified when that version was *active* or *inherited* from its *ancestors*. To instantiate a version, a `materialize` procedure retraces the path from the current version through its *ancestors*, collecting the latest *equivalent operation* on each instance. If the *equivalent operation* is not a `deletion` then it is included in the version description. By maintaining intermediate *complete* versions, a `materialize` operation needs to trace only up to the most immediate *complete ancestor* version which contains a copy of all the changes that could be potentially *inherited* by the version being `materialized`. Figure 3.13 describes the procedure to `materialize` a version that is represented using this modified scheme. Figure 3.14 describes *active* version m-2 of our "ongoing" BEAM example, demonstrating the `materialize` operation in an ORACLE environment. Figure 3.10 in Section 3.4.1 had earlier given the BEAM-DATA relation that described the example version hierarchy.

### 3.5.2 Computing Changes Using a *check-out/check-back-deltas* Protocol

Using a *check-out/check-back-deltas* protocol, we determine the net changes made during a particular session by summarizing the sequence of operations on instances that were made during that session. Two aspects of our model make this feasible. First, we abstract each application operation on an instance into its corresponding primitive operator. Second, we store, as a sequence, all the primitive operators executed on each instance of a given primitive entity that were made during the given session. At the end of this session, a `compress` operator collapses the sequence of operators for each instance into a single root operator, which corresponds to the descriptor of the *equivalent operation* on the particular

instance. By the results of Theorem 1, the `compress` operation is guaranteed to produce a unique and valid *equivalent operation*, provided the original sequence of changes is also valid.

The actual mechanism for collapsing the operator sequence associated with a particular instance is as follows:

1. We substitute the first two nodes of an operator sequence with its equivalent operator that is obtained by the rules given in Section 3.3.1 (implemented as a `get-eqchange` function). Table 3.9 summarizes the `get-eqchange` function for different cases of the first ($u$) and the second ($v$) operators.

2. We repeat Step 1 till the given operator sequence has been reduced to a single node, i.e., there no longer exists a node in the operator sequence that is adjacent to the root node. This resulting root node is the equivalent operator on the given instance during the particular session.

3. We use the results of Corollary 1 (Section 3.3.3) to describe the detected *equivalent operations*. By this result, inserted or replacement values correspond to the instance descriptions at the end of the session, while the replaced or deleted values are originally *checked-out* from the version hierarchy.

Figure 3.15 outlines a `compress` procedure. We use a `query-tuple` function to describe instances logically contained in the *active* version. Similar to a `materialize` operation (presented in Section 3.4.2), this function also retraces the derivation path from the given version till the most recent *complete ancestor* version. However, unlike a `materialize` operation, a `query-tuple` operation terminates when a description of the concerned instance is located; only in the worst case does it retrace upto the most immediate *complete ancestor* version. Figure 3.16 describes an algorithm for a `query-tuple` operator. We consider the following example to demonstrate a `compress` operation. The initial *checked-out* description of version m-2 was given in Figure 3.14. To this description, we execute an example sequence of data operations that is maintained in Table 3.10. The table associates with each modified BEAM instance, the sequence of operators executed on the instance during the considered session. Following are the execution steps of the `compress` operation on each BEAM instance:

1. Instance: Beam 11;

   Iteration 1: *Root*: `replace`;

Table 3.9: Function (`get-eqchange` ($u$, $v$)) for Computing an Equivalent Operation for a Pair of Operators

| descriptor of the first change $u$ $op - desc'$ | descriptor of the second change $v$ *op-desc* | | |
|---|---|---|---|
| | *op-desc* := "insert" | *op-desc* := "delete" | *op-desc* := "replace" |
| $op - desc'$ := "insert" | `print` error; | no operation | `insert;` |
| $op - desc'$ := "delete" | `replace;` | `print` error; | `print` error; |
| $op - desc'$ := "replace" | `print` error; | `delete;` | `replace;` |

Iteration 1: *Tail[0]*: Null;

Descriptor of the *Equivalent Operation*: `replace`

*Equivalent Operation* on Beam 11: `replace` BEAM-DATA(*11*, *18*, *25*) with BEAM-DATA(*11*, *20*, *30*).

2. Instance: Beam 12;

   Iteration 1: *Root*: `delete;`

   Iteration 1:*Tail[0]*: `insert;`

   Iteration 2: *Root*: `replace;`

   Iteration 2: *Tail[0]*: `replace;`

   Iteration 3: *Root*: `replace;`

   Iteration 3: *Tail[0]*: Null;

   Descriptor of the *Equivalent Operation*: `replace`

   *Equivalent Operation* on Beam 12: `replace` BEAM-DATA(*12*, *22*, *35*) with BEAM-DATA(*12*, *20*, *30*).

3. Instance: Beam 14;

   Iteration 1: *Root*: `replace;`

   Iteration 1: *Tail[0]*: `replace;`

   Iteration 2: *Root*: `replace;`

   Iteration 2: *Tail[0]*: Null;

Table 3.10: Example Sequence of Operators on Instances of the BEAM Entity

| *Beam-id* | Sequence of Operator Descriptors |
|---|---|
| 11 | `replace` |
| 12 | `delete` ⇝ `insert` ⇝ `replace` |
| 14 | `replace` ⇝ `replace` |
| 15 | `replace` |

Table 3.11: Example Detected *Equivalent Operations* on *Active* Version (m-2)

| *Change-no* | *Beam-id* | *Wt* | *Span* | *Op-desc* |
|---|---|---|---|---|
| 1 | 11 | 20t | 30ft | "replace" |
| 2 | 12 | 20t | 30ft | "replace" |
| 3 | 14 | 20t | 30ft | "replace" |
| 4 | 15 | 20t | 30ft | "replace" |

Descriptor of the *Equivalent Operation*: `replace`

*Equivalent Operation*on Beam 14: `replace` BEAM-DATA(*14*, *22*, *35*) with BEAM-DATA(*14*, *20*, *30*).

4. Instance: Beam 15;

Iteration 1: *Root*: `replace`;

Iteration 1: *Tail[0]*: Null;

Descriptor of the *Equivalent Operation*: `replace`

*Equivalent Operation*on Beam 15: `replace` BEAM-DATA(15, 18, 25) with BEAM-DATA(15, 20, 30).

Table 3.11 summarizes the net changes on the BEAM instances. These detected *equivalent operations* represent the net changes made during the example application session. Ideally, a `compress` operator would be implemented within an application environment; the current implementation would be inconvenient in situations that have a number of different design applications, some of which might even be customized.

### 3.5.3 Integrating Application Changes with an *Active* Version

An `integrate` operation merges the net design changes on an entity that are *checked-back* with an existing description of the entity's *active* version. For the "optimized" scheme, a particular change has a *matching tuple* if there is an instance in the `materialized` description of the *active* version that has identical primary attribute values as the specified change. There are two possible situations for a change to have a *matching tuple*:

- A tuple with the same primary attribute values is physically associated with the identifier of the *active* version.
- A tuple with the same primary attribute values is logically *inherited* by the *active* version.

We call the former tuple a *physical matching tuple*, the latter is a *logical matching tuple*. Formally,

- A specific change A($\underline{x}$, $y$, $o$) on an instance of entity E has a *physical matching tuple*, if there exists a tuple in the *active* version (the *Version-id* attribute is assigned a value, *active-id*) of the relation E-DATA (represented by E-DATA(*<u>active-id</u>*, $\underline{x}$, $y'$)) that has the same primary attribute values, $x$, as the change A.
- A specific change A($\underline{x}$, $y$, $o$) on an instance of entity E has a *logical matching tuple*, if there exists a tuple in the relation E-DATA (i.e., E-DATA(*<u>version-id</u>*, $\underline{x}$, $y'$)) that has the same primary attribute values, $x$, as the change A and is *inherited* by the *active* version.

We employ the `query-tuple` operator on the *active* version for obtaining the *matching tuple*, if any exists, for a particular application change. If the *matching tuple* is associated with the *active* version itself then it is *physically matching*; the tuple is a *logical matching tuple*, if it is physically associated with an *ancestor* of the *active* version.

We specify two criteria for `integrating` a change into an *active* version. The criteria have been adapted from previous work on view update translation [22].

- **The change is valid in the context of the *active* version.** We can `insert` an instance only if it is not logically contained in the *active* version; Deleted or replaced instances must logically be part of the *active* version.

- **The new change is merged with the current description of the *active* version.** Four factors are proposed to effectively *check-back* a change into an *active* version:
  - *Changes cannot be simpler:* Two or more changes on a given instance are replaced by its *equivalent operation.* Furthermore, an individual change contains all of the attributes in the entity scheme, even if only one attribute is modified; no proper subset of an instance description can be maintained in a version.
  - *No database "side effects":* An entity can only be modified by a change that alters the contents of its *active* version.
  - *No delete - insert pairs:* A modification to one or more non-primary attributes of an instance is referenced as a `replace` operation. Thus, a `delete-insert` pair of operations on an instance with identical primary attribute values is summarized as a `replace` operation.
  - *Key replacements are a delete - insert pair:* Replacement of one or more primary attribute values is represented as a `deletion` of an instance list with the old primary attribute values followed by an `insertion` of another instance list with the new primary attribute values.

These criteria provide the framework for an `integrate` operation. Figure 3.17 outlines a procedure to implement an `integrate` operator. For each *checked-back* change, we invoke an `integrate-change` operator that computes the *equivalent operation* of the *checked-back* change and the description of the same instance that occurs in the *active* version. This computed equivalent operation replaces the current description of the instance in the *active* version. Specific execution of this `integrate-change` operator however depends on the description of the *checked-back* change, as well as any existing *matching tuple.* Table 3.12 summarizes this function for different operator combinations. For each instance, this function determines the *equivalent operation* of its initial description and the sequence of changes on that instance. This resulting *equivalent operation* is now contained in the *active* version. We illustrate an `integrate` operation by modifying the description of *active* version m-2 of the BEAM entity (shown in Figure 3.14) by the *equivalent operations* detected in Table 3.11. These example *equivalent operations* were originally computed earlier in Section 3.5.2 by a `compress` operation using a *check-out/check-back-deltas* protocol. The execution of the `integrate` operation is as follows:

1. Change 1:

*Physical Matching Tuple* does not exist;

*Logical Matching Tuple* exists, *Version-id* := m-0;

`insert` tuple BEAM-DATA(m-2, 11, 20, 30, "replace", m-0);

2. Change 2:

   *Physical Matching Tuple* does not exist;

   *Logical Matching Tuple* exists, *Version-id* := m-1;

   `insert` tuple BEAM-DATA(m-2, 12, 20, 30, "replace", m-1);

3. Change 3:

   *Physical Matching Tuple* does not exist;

   *Logical Matching Tuple* exists, *Version-id* := m-1;

   `insert` tuple BEAM-DATA(m-2, 14, 20, 30, "replace". m-1);

4. Change 4:

   *Physical Matching Tuple* does not exist;

   *Logical Matching Tuple* exists, *Version-id* := m-0;

   `insert` tuple BEAM-DATA(m-2, 15, 20, 30, "replace", m-0)

The final description of the BEAM-DATA relation according to the optimized representation scheme is shown in Figure 3.18. A number of application sessions typically *check-back* their changes into the version hierarchy while a given version is *active*. The `integrate` operation ensures that a version is represented by the set of *equivalent operations* of all changes that were *checked-back* by application sessions, while that version was *active*; the versioning scheme is therefore a *forward deltas* scheme [36]; a version can be described by executing its associated set of *equivalent operations* on the description of its parent version. Furthermore, the version is a unit of granularity whose consistency can be evaluated.

### 3.5.4 Computing changes between versions

A `compute` operation determines the differences between two versions, where one is an *ancestor* of the other in an entity derivation hierarchy. These `computed` changes are a

minimal set of data operations or *deltas* which can be executed on the *ancestor* version to produce a description of the *descendant*.

Figure 3.19 outlines the `compute` procedure. The proposed procedure is an innovative application of *equivalent operations*. Our model views a derivation path between an *ancestor-descendant* pair of versions as a set of sequences of *equivalent operations*, one for each instance that was modified in one or more versions along the derivation path. By this interpretation, a `compute` operation involves determining the net *equivalent operation* for the sequence of *equivalent operations* on each modified instance. Furthermore, as a result of Theorem 2, we can uniquely determine the net *equivalent operation* or *delta* on an instance by considering only the first and last elements of its *equivalent operator* sequence. In other words, for each modified instance, the algorithm does not require the *equivalent operations* from other intermediate versions. This is computationally efficient in design situations having large version hierarchies with a number of intermediate descriptions of an incrementally refined artifact.

Given the first ($u$) and last ($v$) changes on a single instance along a derivation path, a `summarize` function computes the net *equivalent operation*. Table 3.13 tabulates the execution of the `summarize` function for different operator assignments of the first and last change. When the the first change (u) is a `replace` operation, we obtain the replaced instance description from the *ancestor* version in which it is stored. This tuple is denoted as $w$ and is located by the *Vprev* attribute associated with the tuple $u$. In addition to, the *external functions* provided in Section 3.4.2, a `compute-deltas` operator uses the following three functions `pathchild`, `length`, and `anc`:

- `pathchild` = (identifier P) identifier V: Retrieves the child version V of the version P along a predefined derivation path between an *ancestor-descendant* pair of versions.
- `length` = ([1:n] primary H) integer K: Determines the number of elements K in a set of primary attribute values H.
- `anc` = (tuple T) $id_{version}$ A: Returns the identifier of the *ancestor* version A where the current instance was most recently modified. This is stored in the attribute, *Vprev*, in the E-DATA relation.

Figure 3.20 demonstrates the execution of a `compute` operator in a relational system, by determining the differences between versions m-0 and m-2 in the BEAM-DATA relation that was previously shown in Figure 3.18.

### 3.5.5 Removing a version

We outline a procedure to `remove` a particular version from an entity derivation hierarchy. When intermediate versions are `removed`, the version hierarchy is reconfigured; children of the *removed* version are now specified as the children of its parent. Additionally, instances physically associated with the *removed* version must be `deleted`. Unfortunately, in our representation scheme, tuples associated with a given version can be logically *inherited* by its successors. These tuples are also used to `compute` differences between two versions, in whose derivation path the *removed* version lies. Thus, any strategy to `delete` the contents of a *removed* version must ensure: (i) descriptions of other versions in the hierarchy are not be affected, and (ii) computation of changes between any two versions must remain unchanged. To satisfy these two restrictions we `merge` the changes associated with the identifier of the *removed* version with the descriptions of each of its child versions. Success of this approach can be reasoned as follows:

1. Descriptions of successor versions are not affected: `Removing` a version could potentially affect the procedure to `materialize` those successor versions which *inherit* tuples from the version being `removed`. Such situations arise when the child of the *removed* version along each of the concerned derivation paths does not contain the instances being *inherited*. By `merging` the contents of the *removed* version with its children, we explicitly copy all such *inherited* tuples into the interested child versions. Successor versions now *inherit* the particular tuples from the child versions; the child versions explicitly associate the tuples with their identifiers and thus do not need to *inherit* them.

2. Computation of changes between any two versions are not affected: We `merge` the description of all instances contained in the version being `removed` version with each of its child versions. Thus, we store with each child version, the *equivalent operations* on all instances that were contained in the *removed* version or the considered child version or both. Additionally, as the changes associated with the version to be `removed` are `merged` with each of its children, corresponding sets of *equivalent operations* are maintained in all possible derivation paths that contain the *removed* version; computation of changes along any derivation path is not affected.

An algorithm to implement a `remove` operation for the proposed "optimized" relational scheme executes the following operations on the E-DATA, E-INDEX, and E-ACTIVE relations:

- Eliminates the concerned version from the E-INDEX relation, and modifies the parent links of its child versions to reconfigure the hierarchy; children of the *removed* version are now children of its parent version. If the *removed* version was in the *active* state, then it is `deleted` from the E-ACTIVE relation as well.

- Merges the tuples associated with the version to be `removed` in the E-DATA relation with the tuples associated with each of its child versions.

Figures 3.21 and 3.22 describe the procedures to `remove` a version from the hierarchy. The procedure in Figure 3.21 modifies the descriptions of the child versions to ensure that the computation of changes is not affected, while Figure 3.22 describes the database operations to reconfigure the hierarchy. These database operations have been presented in a syntax similar to SQL [41]. In addition to the *external functions* that are provided in Section 3.5.4, the `remove` algorithms use the following functions:

- `delete` = ($id_{version}$ V, primary X): Eliminates from the relation E-DATA the tuple identified by the version identifier V and the primary attributes X.
- `next-child` = ($id_{version}$ P) $id_{version}$ C: Obtains the child version C of the version P that is subsequent to one that was most recently referenced, as determined by the current position of a *Vers-locator* pointer. Successful execution of a `next-child` function advances the pointer by one component.

Table 3.14 describes the procedure to substitute an existing change in the child version with the *equivalent operation* of the instance in the parent *removed* version ($u$) and the child version ($v$). The execution of a `merge` function for different cases of changes in the parent and child versions are provided by the rules specified in Section 3.3.1. Table 3.15, on the other hand, considers situations where the child version does not contain a tuple on the instance being `merged`. In situations where the concerned tuple is one of `insert`, `delete` or `replace` operations) the tuple is physically `inserted` into the child version. Further, the *Vprev* attribute value of a version that initially pointed towards the *removed* version is now redirected to its child version that lies on the path from the given version to the *removed* version. However, if the considered is a `copy` operation, it need not be `merged` into the child version. Also, in this case the *Vprev* attribute value in the successor version is redirected

to the ancestor version from which the tuple was originally copied. In every situation the tuples associated with the version being *removed* are `deleted` from the E-DATA relation.

**Algorithm 9** (`materialize` - **Optimized Model**)

*Input:* *e:* Name of the concerned entity.
*materialize-id*: Identifier of the version being `materialized`.

*Output:* tuples that are logically part of the version that is materialized.

*External Function:* the function `instantiate-tuple` is defined in Algorithm 8.

```
procedure materialize = (entity e, id_version materialize-id):
begin id_version version-id, tuple u, primary x, secondary y;
  version-id := materialize-id;
  while (version-id ≠ "Null" AND vers-type(version-id) ≠ "complete") do
  begin
    u:= retrieve(version-id);
    while (u ≠ EOF) do
    begin
      success:= instantiate-tuple(version-id, u);
      x:= key(u);
      y:= dependent(u);
      if (success) begin
        output (π_{x,y} u);
      end;
      u:= retrieve(version-id);
    end;
    version-id := vers-parent(version-id);
  end;
  u:= retrieve(version-id);
  while (u ≠ EOF) do
  begin
    success:= instantiate-tuple(version-id, u);
    x:= key(u);
    y:= dependent(u);
    if (success) begin
      output (π_{x,y} u);
    end;
    u:= retrieve(version-id);
  end;
end procedure
```
◇

Figure 3.13: Procedure to `materialize` a Version of a Primitive Entity Represented by the "Optimized" Scheme

```
SQL> select * from Beam_material;

   BEAM_ID         WT       SPAN
---------- ---------- ----------
        14         22         35
        13         16         20
        12         22         35
        15         18         25
        11         18         25

SQL> ▯
```

Figure 3.14: Relational Description of `materialized` BEAM Version m-2

Table 3.12: The (`integrate-change`) Operator to Modify a Version Description.

| Status of *matching tuple* in E-DATA | Change A | | |
|---|---|---|---|
| | A($\underline{x}$, $y'$, "insert") | A($\underline{x}$, $y'$, "delete") AND ($y = y'$) | A($\underline{x}$, $y'$, "replace") |
| *logical matching tuple* exists *tuple* exists | `print` error; | `insert` tuple E-DATA(*active-id*, $\underline{x}$, $y'$, "delete", *anc-id*) | `insert` tuple E-DATA(*active-id*, $\underline{x}$, $y'$, "replace", *anc-id*) |
| not existent, also no *logical matching tuple* exists | `insert` tuple E-DATA(*active-id*, $\underline{x}$, $y'$, "insert", "Null") | `print` error; | `print` error; |
| *physical matching tuple* exists E-DATA(*active-id*, $\underline{x}$, $y$, "insert", "Null") | `print` error; | `delete` matching tuple; | `replace` tuple to E-DATA(*active-id*, $y'$, "insert", "Null"); |
| E-DATA(*active-id*, $\underline{x}$, $y$, "delete", a) | `replace` tuple to E-DATA(*active-id*, $\underline{x}$, $y'$, "replace", a); | `print` error; | `print` error; |
| E-DATA(*active-id*, $\underline{x}$, $y$, "replace", a) | `print` error; | `replace` tuple to E-DATA(*active-id*, $\underline{x}$, $y'$, "delete", a); | `replace` tuple to E-DATA(*active-id*, $\underline{x}$, $y'$, "replace", a); |
| E-DATA(*active-id*, $\underline{x}$, $y$, "copy", a) | `print` error; | `replace` tuple to E-DATA(*active-id*, $\underline{x}$, $y'$, "delete", a); | `replace` tuple to E-DATA(*active-id*, $\underline{x}$, $y'$, "replace", a); |

**Algorithm 10** (compress)

*Input:* *g:* is the number of instances that were modified in that application session
*op* is the sequence of *n* changes on each instance of the entity E-DATA.
*t* is the description of each instance of the entity E at the end of the application session.

*Output:* Equivalent operation on each instance that was modified in the particular application session.

```
procedure compress-instance = (op-desc root, [1:t] op-desc tail, entity e) op-desc
    root:
begin integer i;
  while (tail[1] ≠ "Null") do
  begin
    root := get-eqchange(root, tail[1]);
    for i := 1 to t-1 do
    begin
      tail[i] := tail[i+1];
    end;
    root := compress-instance(root, tail, E);
  end;
end procedure
procedure compress = ([1:g][1:n] op-desc op, [1:g] tuple t, entity e):
begin integer k, j; op-desc root; [1:n-1] op-desc tail;
  for k := 1 to g do
  begin
    root := op[k][1];
    for j := 1 to n-1 do
      tail[j] := op[k][j+1];
    end;
    root := compress-instance(root, tail, e);
    if (root = "insert") begin
      output(t[k], "insert");
    end;
    if (root = "delete") begin
      output(query-tuple(active-id, key(t[k])), "delete");
    end;
    if (root = "replace") begin
      if (NOT compare(query-tuple(active-id, key(t[k])), T[k])) begin
        output(query-tuple(active-id, key(t[k])) → t[k], "replace");
      end;
    end;
  end;
end procedure
◇
```

Figure 3.15: Algorithm to compress a Sequence of Operators on Each Modified Instance to Detect the Net Changes Made During an Application Session

**Algorithm 11** (query-tuple)

*Input:* *current-id*: Identifier of the version for which the existence of a matching tuple is to be determined.

$x$: Values assigned to the primary attributes of the instance of the entity E that is being queried.

*Output:* *success-id*: identifier of the version in which a description of the queried instance is located.

```
procedure query-tuple = (id_version current-id, primary instance-id) id_version
      success-id:
begin id_version version-id, tuple u, primary x, secondary y, boolean success;
   version-id := current-id;
   while (version-id ≠ "Null" AND vers-type(version-id) ≠ "complete" AND (NOT success)) do
   begin
      u:= fetch(version-id, x);
      while (u ≠ EOF AND (NOT success)) do
      begin
         success := TRUE;
         if (operator(u) ≠ "delete") begin
            output (π_{x,y} u);
            success-id := version-id;
         end;
      end;
      version-id := vers-parent(version-id);
   end;
   if (version-id ≠ "Null" AND (NOT success)) begin
      u:= fetch(version-id, x);
      while (u ≠ EOF AND (NOT success)) do
      begin
         success := TRUE;
         if (operator(u) ≠ "delete") begin
            output (π_{x,y} u);
            success-id := version-id;
         end;
      end;
   end;
end procedure
◇
```

Figure 3.16: Procedure to Query a Version for an Instance Logically Belonging to It: query-tuple Operator

**Algorithm 12** (`integrate`)

*Input:* *e:* Name of the concerned entity.

*ch:* Set of $g$ sequences of changes on the entity E-DATA, each element of which is a valid sequence of $n$ data operations on a specific instance of the entity E identified by the values assigned to the key attributes $X$.

*g:* Number of instances that were modified in that application transaction.

*n:* Number of changes on each instance.

*Output:* E-DATA with the summary of changes committed

```
procedure integrate-instance = ([1:n] change ch, entity e):
begin integer i;
  for i := 1 to n do
  begin
    integrate-change(ch[i], e);
  end;
end procedure


procedure integrate = ([1:g][1:n] change ch, entity e):
begin integer k;
  for k := 1 to g do
  begin
    integrate-instance(ch[k], e);
  end;
end procedure
◇
```

Figure 3.17: Procedure to `integrate` Application Changes on an Entity with its *Active* Version

```
SQL> select * from Beam_Data
  2  ;

VERSION_ID  BEAM_ID         WT       SPAN OP_DESC   VPREV
---------- ---------- ---------- ---------- ---------- ----------
m-0                11         18         25 insert     Null
m-0                12         20         30 insert     Null
m-0                13         20         30 insert     Null
m-0                14         20         30 insert     Null
m-0                15         18         25 insert     Null
m-0a0              11         20         30 replace    m-0
m-0a0              12         18         25 replace    m-0
m-0a0              14         18         25 replace    m-0
m-0a0              15         20         30 replace    m-0
m-1                12         22         35 replace    m-0
m-1                13         16         20 replace    m-0

VERSION_ID  BEAM_ID         WT       SPAN OP_DESC   VPREV
---------- ---------- ---------- ---------- ---------- ----------
m-1                14         22         35 replace    m-0
m-2                11         20         30 replace    m-0
m-2                12         20         30 replace    m-1
m-2                14         20         30 replace    m-1
m-2                15         20         30 replace    m-0

16 rows selected.

SQL> 
```

Figure 3.18: Description of the BEAM Entity after Modifying *Active* Version m-2

Table 3.13: `summarize` $(u, v, w)$ Function for a Sequence of Changes on an Instance

| first change | last change $v :=$ E-DATA(*last-id*, x, $y'$, *op-desc*, a) | | | |
|---|---|---|---|---|
| | *op-desc* := "copy" | *op-desc* := "insert" | *op-desc* := "delete" | *op-desc* := "replace" |
| $u :=$ E-DATA(*first-id*, x, $y$, "copy", *anc-id*) | if($y \neq y'$) `replace` $(u \rightarrow v)$; | `replace` $(u \rightarrow v)$; | `delete`$(u)$; | `replace` $(u \rightarrow v)$; |
| $u :=$ E-DATA(*first-id*, x, $y$, "insert", "Null") | `insert`$(v)$; | `insert`$(v)$; | nop | `insert`$(v)$; |
| $u :=$ E-DATA(*first-id*, x, $y$, "delete", *anc-id*) | `replace` $(u \rightarrow v)$; | `replace` $(u \rightarrow v)$; | `delete`$(u)$; | `replace` $(u \rightarrow v)$; |
| $u :=$ E-DATA(*first-id*, x, $y$, "replace", *anc-id*) | `replace` $(w \rightarrow v)$; | `replace` $(w \rightarrow v)$; | `delete`$(w)$; | `replace` $(w \rightarrow v)$; |

Note: $w :=$ E-DATA(*anc-id*, x, $y''$, *op−desc*$'$, $a'$)

**Algorithm 13** (compute)

*Input:* $version_A$: identifier of the ancestor version (ancestor of $version_B$) in version hierarchy of entity E

$version_B$: identifier of the descendant version (descendant of $version_A$) in version hierarchy of entity E

*first*: first data operation on each instance in the derivation path from $version_A$ to $version_B$

*last*: last data operation on an instance in the derivation path from $version_A$ to $version_B$

*anc*: ancestor of the first data operation on each instance in the derivation path from $version_A$ to $version_B$.

*Output:* *eq*: equivalent data operation on each instance that is modified in the derivation path from $version_A$ to $version_B$.

```
procedure range = (id_version range-id, [1:n] tuple first, last, initial, [1:n] primary h)
              [1:n] tuple first, last, initial, [1:n] primary h:
begin tuple u; primary x; secondary y; id_version version-id;
  version-id := range-id;
  u := retrieve(version-id);
  while (u ≠ EOF) do
  begin
    if ((x := key(u) ∉ h) begin
      add(x, h);
      if (operator(u) = "replace") begin
        anc-id := anc(u);
        initial[x] := fetch(anc-id, x);
      end;
      first[x] := u;
      last[x] := u;
    end;
    else begin
      last[x] := u;
    end;
    u := retrieve(version-id);
  end;
end procedure
```

◇

**Algorithm 13 (contd.)** `compute`

```
procedure compute = (id_version versionA, versionB) [1:n] tuple eq:
begin id_version version-id; [1:n] tuple first, last, initial; [1:n] primary h; integer i;
   for i := 1 by 1 to n begin
      h[i] := 0;
   end;
   version-id := pathchild(versionA);
   while (version-id ≠ "Null" AND version-id ≠ versionB) do
   begin
      range(version-id, first, last, initial, h);
      version-id := pathchild(version-id);
   end;
   range(version-id, first, last, initial, h);
   for i := 1 by 1 to length(h) begin
      eq[h[i]] := summarize(first[h[i]], last[h[i]], initial[h[i]]);
   end;
end procedure
◇
```

Figure 3.19: Algorithm to `compute` the Changes Between Two Versions Where One Version is an *Ancestor* of the Other

```
SQL> select * from Beam_rchanges;

   BEAM_ID         WT       SPAN OP_DESC
---------- ---------- ---------- ----------
        11         18         25 replaced
        11         20         30 replacemt
        13         20         30 replaced
        13         16         20 replacemt
        15         18         25 replaced
        15         20         30 replacemt

6 rows selected.

SQL> 
```

Figure 3.20: Relational Representation of Changes Between Two Versions

**Algorithm 14** (remove)

*Input:* *e:* Name of the concerned entity.

*remove-id*: Identifier of the version being removed

*Output:* E-DATA and E-INDEX relations without the version, *remove-id*.

```
procedure remove = (entity e, id_version remove-id):
begin tuple u, v, w; primary x; secondary y, y'; [1:n] tuple z; [1:n] primary h;
                 id_version version-id; integer k;
   u := retrieve(remove-id);
   while (u ≠ EOF) do
   begin
      x := key(u);
      add (x, h);
      add (u, z);
      u:= retrieve(remove-id);
   end;
   version-id := nextchild(remove-id);
   while (version-id ≠ "Null") do
   begin
      for (i := 0 by 1 to length(h))
      begin
         v := fetch(version-id, x);
         merge(u, v);
      end;
      version-id := nextchild(remove-id);
   end;
   delete(remove-id, e);
end procedure
◇
```

Figure 3.21: Data Operations to remove a Version from an Entity Derivation Hierarchy

**Algorithm 15 (Remove Version: Pruning the Hierarchy)**

UPDATE E-INDEX
SET *Parent-id* = parent(*remove-id*))
WHERE *Version-id* = SELECT *Version-id*
FROM E-INDEX
WHERE *Parent-id* = *remove-id*

DELETE FROM E-INDEX
WHERE *Version-id* = *remove-id*
◇

Figure 3.22: Reconfiguring a Version Hierarchy When `removing` a Version Definition

Table 3.14: Function, `merge`(*remove-id*), to Merge Changes Between the *Removed* Version and its Children

| E-DATA(*Version-id*, *X*, *Y*, *Op-desc*, *Vprev*) | | | | |
|---|---|---|---|---|
| tuple in parent version being `removed` | existing tuple in child version *v* := E-DATA(*child-id*, *x*, *y′*, *op-desc*, *remove-id*) | | | |
| | *op-desc* := "copy" | *op-desc* := "insert" | *op-desc* := "delete" | *op-desc* := "replace" |
| *u* := E-DATA (*remove-id*, *x*, *y*, "copy", *anc-id*) | `update` *v* `set` *Vprev* := *anc-id*; | `print` error; | `update` *v* `set` *Vprev* := *anc-id*; | `update` *v* `set` *Vprev* := *anc-id*; |
| *u* := E-DATA (*remove-id*, *x*, *y*, "insert", "Null") | `update` *v* `set` *Vprev* := "Null" AND `set` *Op-desc* := "insert"; | `print` error; | `delete`(*v*); | `update` *v* `set` *Vprev* := "Null" AND `set` *Op-desc* := "insert"; |
| *u* := E-DATA (*remove-id*, *x*, *y*, "delete", *anc-id*) | `print` error; | `update` *v* `set` *Vprev* := *anc-id* AND `set` *Op-desc* := "replace"; | `print` error; | `print` error; |
| *u* := E-DATA (*remove-id*, *x*, *y*, "replace", *anc-id*) | `update` *v* `set` *Vprev* := *anc-id* AND `set` *Op-desc* := "replace"; | `print` error; | `update` *v* `set` *Vprev* := *anc-id*; | `update` *v* `set` *Vprev* := *anc-id*; |

Table 3.15: Merging of Changes (Contd.) (`merge`)

| tuple in parent version | tuple not existent tuple in child version |
|---|---|
| E-DATA(<u>*remove-id*</u>, <u>*x*</u>, *y*, "*copy*", *anc-id*) | *w* := E-DATA(<u>*version-id*</u>, <u>*x*</u>, *y''*, *op-desc*, *remove-id*)<br>`update` *w*<br>`set` *Vprev* := *anc-id*; |
| E-DATA(<u>*remove-id*</u>, <u>*x*</u>, *y*, "*insert*", "*Null*") | *w* := E-DATA(<u>*version-id*</u>, <u>*x*</u>, *y''*, *op-desc*, *remove-id*)<br>`update` *w*<br>`set` *Vprev* := *child-id*;<br>`insert`<br>E-DATA(<u>*child-id*</u>, <u>*x*</u>, *y*, "insert", "Null"); |
| E-DATA(<u>*remove-id*</u>, <u>*x*</u>, *y*, "*delete*", *anc-id*) | *w* := E-DATA(<u>*version-id*</u>, <u>*x*</u>, *y''*, *op-desc*, *remove-id*)<br>`update` *w*<br>`set` *Vprev* := *child-id*;<br>`insert`<br>E-DATA(<u>*child-id*</u>, <u>*x*</u>, *y*, "delete", *anc-id*); |
| E-DATA(<u>*remove-id*</u>, <u>*x*</u>, *y*, "*replace*", *anc-id*) | *w* := E-DATA(<u>*version-id*</u>, <u>*x*</u>, *y''*, *op-desc*, *remove-id*)<br>`update` *w*<br>`set` *Vprev* := *child-id*;<br>`insert`<br>E-DATA(<u>*child-id*</u>, <u>*x*</u>, *y*, "replace", *anc-id*); |

## 3.6 Summary and Discussions

This chapter described a version model to manage evolving descriptions of a primitive entity in a single discipline. Specifically, we maintain the version set as a tree structure in which branching allows multiple alternatives to be developed in parallel. Each version in a particular hierarchy contains specific descriptions of instances of that entity.

Research efforts in the software engineering, CAD, and database communities have proposed to managing evolving descriptions of data items [18]. In the design environment, various structures, including the hierarchy [19] and the DAG (directed acyclic graph) [25] have been proposed to organize individual versions. Although a DAG is a more general structure, we are aware of no general algorithms that detect and resolve conflicts that arise when version descriptions are merged.

Efficient storage and retrieval of changes is critical to supporting an evolutionary process. Starting with the Source Code Control System (SCCS) model [36], the software engineering community has explored maintaining versions as a set of *deltas* or data operations on the contents of the version. A number of variations have been proposed on storing the *delta* set. Using a *forward* delta set, a version is described by executing its delta set on its parent version's description. A *reverse* delta set is just the opposite, executing the delta set associated with a version on its description results in the description of its parent version. The SCCS [36] model is based on storing *forward* deltas scheme. Each delta, in this model, is either an `insertion`, `deletion`, or `replacement` of a line of text. A version is described by applying the deltas sequentially from the root till the specified version. To manage a version chain, SCCS partitions it into levels, distinguishing releases within each level. In a particular level, a new delta set can only be added to the end of the last release. The RCS model [40] extends the SCCS model by supporting a more general branching mechanism. To improve access time, the most recent version on the trunk is stored intact. All other revisions on the trunk are stored as *reverse* deltas. Thus, the extraction of the latest version is a simple fast copy operation. Earlier versions are described by applying the reverse deltas to the latest version. The model uses *forward* deltas to store branches. Regenerating a revision on a side branch involves extracting the latest revision on the trunk, applying reverse deltas until the revision that forks into the desired branch, and then applying the forward deltas until the branch version of interest is reached. Although the RCS model is efficient in accessing the latest version in the trunk, it performs poorly in materializing branch versions in situations

where branching occurs early in the hierarchy. Unfortunately, in design scenarios branching typically occurs early in the process; a designer usually generates multiple alternatives in the conceptual phase, and refines one or more of these alternatives through the design process. Also, the SCCS and RCS models suffer from the following shortcomings:

1. They have not provided any systematic criteria to integrate application changes with an existing version description.

2. They have not specified algorithms to compute changes between versions, where one has evolved from the other.

3. They have not addressed implications of pruning a version hierarchy by removing intermediate versions. Removing a version should not affect the instantiation of its descendant versions. Also the computation of changes between two versions must remain unaffected on removing one or more intermediate versions along their derivation path.

Although Spooner et al. [37] have developed rules to merge changes on an instance from multiple concurrent transactions, they have not addressed the integration of the merged change set with an existing version description. Neither are we aware of any research efforts that have studied the complementary problem of detecting changes made to structured design representations in CAD application environments. Algorithms, similar to UNIX's diff program [16], have been utilized to compute changes between two versions by comparing their descriptions. Given that design versions are represented by more structured entity schemes, we can improve on diff type algorithms by applying more sophisticated techniques than a simple scan of two files to search for matching lines of text.

We apply a concept of *equivalent operations* to store, detect and manage changes among versions in an entity hierarchy. Intuitively, an *equivalent operation* for a sequence of changes results in the same final description of the instance as the original sequence of changes. Using the concept of *equivalent operations*, we establish a version as the set of *equivalent operations* of all changes on that entity that were made while the given version was *active*. Thus, the versioning scheme is a *forward deltas* scheme, a version can be described by executing its associated set of changes (set of *deltas*) on the description of its parent version. For greater computational efficiency, we store instantiated descriptions of certain intermediate versions. Such instantiated versions are denoted as *complete*; the remaining versions are *incomplete*.

A `materialize` operator describes a version by retracing the path from the current version through its *ancestors*, collecting the latest *equivalent operation* on each instance. If the *equivalent operation* is not a `delete` operation, it is included in the version description. By maintaining intermediate *complete* versions, a `materialize` operation needs to trace only up to the most immediate *complete* ancestor version which contains a copy of all changes that could be potentially *inherited* by the version being `materialized`.

We introduce a *check-out/check-back-deltas* protocol to structure the interaction between an application session and the *active* version. To detect the net changes made during a session, we `compress` the sequence of operators on each instance that was modified during that session. Two aspects of our model make this feasible. First, we abstract each application operation on an instance into its corresponding primitive operator. Second, we maintain as a sequence the descriptors of all operations that were executed on each modified instance. The detected *equivalent operation* on each instance is then `integrated` with the existing description of the concerned entity's version. Since an `integrate` operation merges the changes from all application sessions that *check-back* their changes into an *active* version, a version is a unit of granularity whose consistency can be evaluated.

A `compute` operation determines the differences between two versions, where one is an *ancestor* of the other in an entity derivation hierarchy. These determined changes are a minimal set of data operations or *deltas* which can be executed on the *ancestor* version to describe its *descendant*. The `compute` operator is an innovative application of *equivalent operations*; the operator determines the net change on each instance from the first and last description of the instance in versions along the derivation path.

Storage considerations prompt procedures to prune version hierarchies by `removing`: (i) alternatives that are no longer part of a design solution, and (ii) certain intermediate checkpointed descriptions that were created early in the design process. We outlined a procedure that `merges` the contents of the version being `removed` with each of its child versions. The resulting *equivalent operations* for a given child version replaces its original description. This procedure ensures that the following procedures are unaffected when `removing` a version: (i) to `materialize` successor versions that *inherit* tuples from the version being `removed`, and (ii) to `compute` changes between two versions along whose derivation path lies the version being `removed`.

# Chapter 4

# Assembly and Configuration Models

This chapter describes assemblies and configurations to represent an evolving multidisciplinary project description in terms of the independent evolution of design descriptions from its participating disciplines. Assemblies, in this model, maintain a complex entity as the result of a composite modeling operation on its components. Assemblies can be either *total* or *partial*. *Total* assemblies describe a complete design in an individual discipline, while a *partial* assembly represents a complex entity that can be aggregated with other complex entities to describe an individual design. Configurations are employed as a framework to represent a multidisciplinary design project in terms of design descriptions from each of the participating disciplines. A designer in an individual discipline creates a configuration by integrating a design from his/her discipline with a design from each of the other disciplines. Complexity arises in typical multidisciplinary design environments as designs from the participating disciplines are not mutually exclusive; they share information such as the spatial arrangements and material properties of the elements belonging to the artifact being designed. Constraints can be used to represent restrictions on an individual design due to decisions made in other disciplines [14]. As *total* assemblies represent designs in individual disciplines, a configuration is formally specified as a set of *total* assemblies, one from each participating discipline, and a set of project constraints.

This chapter also demonstrates the change management capabilities of the proposed model. As described in the previous chapter, *equivalent operations* provide the theoretical

foundations to manage changes at the version level. The close coupling of the version, assembly and configuration levels, enables these computed version changes to be recursively combined to represent changes at the various assembly and configuration levels. These concepts are applied to support **project coordination** through *asynchronous communication* of changes among designers, as well as **project monitoring** through systematic tracking of evolving project descriptions. We also propose representation schemes for the assembly and configuration models in a relational environment and discuss prototypical implementations in an ORACLE database system.

## 4.1 Assembly Model for Design Applications

This section presents an assembly model to represent complex entities in an individual discipline, as well as to support collaboration among designers on a project. We first present an overview of the assembly model, discussing the proposed assembly states and operators to specify them. Second, we enumerate access and status properties for assemblies which support the design process. These properties allow designers to share information, both within and outside the design team, while retaining control over the descriptions for which they provide access. Also, assemblies can be maintained over extended periods, often covering the facility's life cycle. Third, we present a scheme for representing assemblies in a relational context. This scheme establishes both *composition* and *evolution* relationships among assemblies. While a *composition* relationship identifies the versions included in an assembly definition, an *evolution* relationship identifies an earlier description of the complex entity from which the current assembly has been generated. Additionally, the scheme maintains the status and access properties assigned for each assembly in a particular discipline. The close coupling of the version and assembly model requires that specific assembly status and access property values are shared by each of its component versions. The fourth part of this section extends the version model given in Chapter 3 (Section 3.1) by providing status and access properties for version definitions; the scheme of an E-INDEX relation given in Section 3.4.3 is extended to store the assigned property values. The sharing of assembly properties by its components is enforced by the assembly state operators. In the fifth part, we formally present the operators included in the assembly model. Finally, the sixth part of this section discusses the management of changes among assemblies along both the *composition* and *evolution* links. Along a *composition* relationship, we instantiate an assembly

as an aggregation of `materialized` descriptions of its component versions. Along an *evolution* relationship, on the other hand, we characterize the changes between two assemblies in terms of `computed` changes between corresponding pairs of component versions. We discuss the implementation of these procedures in a relational database environment.

### 4.1.1 Overview of the Assembly Model

An assembly, in our model, represents a complex entity as a result of a composite modeling operation on a set of component instances. Based on its existence, we classify an assembly definition into one of the following two states:

- *Defined assembly*, which describes an instance of a complex entity as the result of a composite modeling operation on its components.

- *Eliminated assembly*, which was previously *defined* but no longer exists.

Figure 4.1 shows graphically the finite state representation of the assembly model. The model provides two alternative approaches to create a new *defined* assembly. A new assembly can be either `defined` independently as a composite modeling operation on its component instances. or `generated` as a *child* of another *defined* assembly. When `generating` a new assembly, a designer substitutes one or more components of the parent assembly with more refined descriptions. The operation thus tracks an evolving description of a complex entity. Further, each component of the parent assembly is guaranteed to be an *ancestor* of the corresponding component in the child assembly. A transitive closure of the parent links establishes *ancestor-descendant* relationships among assemblies. An assembly is therefore a *proper ancestor* of another, if the first assembly is either a *parent* or a *proper ancestor* of the parent of the second assembly. Relaxing the restriction of a proper ancestor, we specify that an assembly is an *ancestor* of itself. The assembly model also permits the `elimination` of additional assemblies to minimize the storage requirements.

Before describing the assembly state operators in detail, we outline the data type declarations used in our discussions:

- discipline: Type declaration of the discipline under consideration.

- $\text{id}_{\text{assembly}}$: Type declaration of the identifier of the specific assembly in a particular discipline.

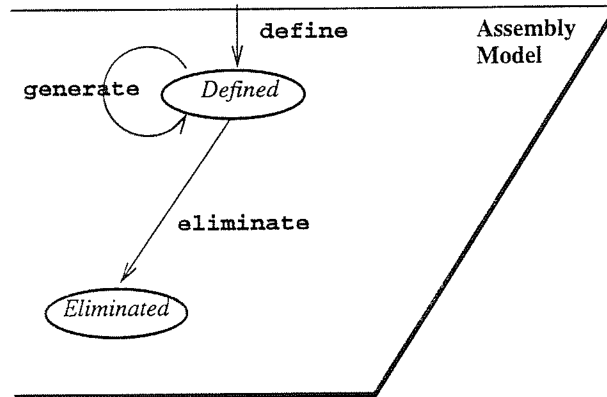



Figure 4.1: Assembly Model as a Finite State Machine

- $\text{state}_{\text{assembly}}$: State specification of an assembly definition.

We employ the following *external functions* to develop algorithms for the proposed assembly state operators:

- `ably-numgen` = (discipline D, [1:n] $\text{id}_{\text{version}}$ V, $\text{id}_{\text{assembly}}$ Q) $\text{id}_{\text{assembly}}$ A: Generates an identifier A for a new assembly in a discipline D that is `generated` as a child of an existing *defined* assembly Q, and has $n$ component versions. When `defining` a new assembly in that discipline, the identifier of the parent version Q is assigned a "Null" value. This function also stores the assembly definition in terms of its components, which can be retrieved by other operators.

- `ably-destroy` = (discipline D, $\text{id}_{\text{assembly}}$ A) boolean Z: Erases the assembly definition A from the set of assemblies in discipline D. Z indicates a successful execution of the operator.

- `ably-parent` = (discipline D, $\text{id}_{\text{assembly}}$ A) $\text{id}_{\text{assembly}}$ Q: Retrieves the identifier of the parent Q of an assembly A in the concerned discipline D. The function returns the identifier of the assembly from which the current assembly has been `generated`. If the assembly A were independently `defined`, then this function returns a "Null" value.

- `ably-state` = (discipline D, $\text{id}_{\text{assembly}}$ A) $\text{state}_{\text{assembly}}$ W: Obtains the current state specification W of an assembly A belonging to a discipline D.

**Algorithm 16** (`define-assembly`)

*Input:* $d$: Name of the discipline in which the assembly is being `defined`.
$v$: Identifier of component versions of the new assembly.
$n$: Total number of entities included in the assembly definition.

*Output:* a: Identifier of the newly *defined* assembly.

**procedure** `define-assembly` = (discipline $d$, [1:n] $\text{id}_{\text{version}}$ $v$) $\text{id}_{\text{assembly}}$ a:
**begin**
  a := `ably-numgen`($d$, [1:n] $v$, "Null");
  `ably-state`($d$, a) := *defined*;
  **end**;
**end procedure**
◇

Figure 4.2: Procedure to `define` a New Assembly in a Particular Discipline

We now describe the assembly operators in detail:

- `define-assembly`(discipline D, [1:n] $\text{id}_{\text{version}}$ V) $\text{id}_{\text{assembly}}$ A: Creates a new assembly in discipline D given a set of $n$ component versions. The assembly is *total* if it contains a version of each entity in that discipline. A `define-assembly` operator initiates a call to a system function `ably-numgen` that generates an identifier A for the assembly. Figure 4.2 outlines an algorithm to implement the `define-assembly` operator.

- `generate-assembly`(discipline D, [1:n] $\text{id}_{\text{version}}$ V, $\text{id}_{\text{assembly}}$ Q) $\text{id}_{\text{assembly}}$ A: Creates a new assembly A in a discipline D and links it to a previously *defined* assembly Q, as a child of that assembly. The operator invokes an `ably-numgen` function to generate the identifier A for the new child assembly. An algorithm for a `generate` operator is presented in Figure 4.3. Each component version in assembly A is guaranteed to be a *descendant* of its corresponding version (version of the same entity) in assembly Q.

- `eliminate-assembly`(discipline D, $\text{id}_{\text{assembly}}$ A): Specifies an existing *defined* assembly A in discipline D as *eliminated*. Figure 4.4 shows an algorithm to `eliminate` an assembly definition from a particular discipline. The algorithm does not `remove` any of the assembly's component versions, as they could be components of other assemblies as well.

**Algorithm 17** (`generate-assembly`)

*Input:* $d$: Name of the discipline in which the assembly is being `generated`.
$q$: Identifier of a previous *defined* assembly, a child of which is being `generated`.
$v$: Set of $n$ component version identifiers. Each component version identifier is guaranteed to be a *descendant* of the corresponding version in assembly $q$.
$n$: Total number of entities included in the assembly definition.

*Output:* a: Identifier of the newly `generated` assembly. The assembly is specified in the *defined* state.

**procedure** `generate-assembly` = (discipline $d$, [1:n] $\text{id}_{\text{version}}$ v, $\text{id}_{\text{assembly}}$ $q$) $\text{id}_{\text{assembly}}$ a:
**begin**
  a := `ably-numgen`($d$, [1:n] v, $q$);
  `ably-state`($d$, a) := *defined*;
  **end**;
**end procedure**
◇

Figure 4.3: Procedure to `generate` a New Assembly as a Child of an Existing *Defined* Assembly

These three operators completely specify the assembly states needed for designing in a particular discipline.

Assemblies can be alternatively classified as *total* or *partial*. A *total* assembly describes a complete design in an individual discipline and contains in its definition at least one version of each entity in that discipline. A *partial* assembly, on the other hand, represents a complex entity in that discipline which can be further combined with other *partial* assemblies to describe more complex entities.

### 4.1.2 Assembly Properties for Collaboration

To collaborate on a project, designers need to share their designs with the remaining team members. An access property, "publish," for a given assembly specifies that designers from other disciplines can reference its contents. However, the model must ensure that an assembly not be `removed` while designers from other disciplines are accessing it. We therefore specify a status property, "freeze," which specifies that a given assembly cannot be explicitly `eliminated` from an entity derivation hierarchy. Since certain assemblies could

**Algorithm 18** (`eliminate-assembly`)

*Input:* *d:* Name of the concerned discipline.
*a:* Identifier of the existing assembly which is being `eliminated`.
*Output:* *success:* Indicator of the a successful operation.

**procedure** `eliminate-assembly` = (discipline *d*, $\text{id}_{\text{assembly}}$ a) boolean *success*:
**begin**
*success* := FALSE;
**if** (`ably-state`(*d*, a) = *defined*) **begin**
`ably-state`(*d*, a) := *eliminated*;
*success* := `ably-destroy`(*d*, a);
**end**;
**end procedure**
◇

Figure 4.4: Procedure to `eliminate` an Existing Assembly from a Particular Discipline

be components of project designs that are maintained for extended periods, often covering the facility's life cycle, they must not be `eliminated` at least during the life cycle of the project. This feature is ensured by a status property, "archive," for assemblies to guarantee their existence.

Based on the specific assignments of status and access property values, we further classify an assembly definition into one of the following four additional states:

- *Frozen assembly,* which cannot be explicitly `removed`; status property value is "freeze." Contents of a *frozen* assembly are not accessible to designers from other disciplines; access value is "not publish."
- *Published assembly,* whose contents can be referenced by designers in other disciplines; access property value is "publish." A *published* assembly cannot be `removed` while being accessed by other designers; status value is "freeze."
- *Archived assembly,* which is guaranteed to exist for the lifetime of the database. An *archived* assembly is not accessible to designers from other disciplines. The assembly has a status value, "archive," and an access value, "not publish."
- *Persistent assembly,* which is guaranteed to exist for the lifetime of the database. At the same time, it is accessible to designers from other disciplines, as well as actors

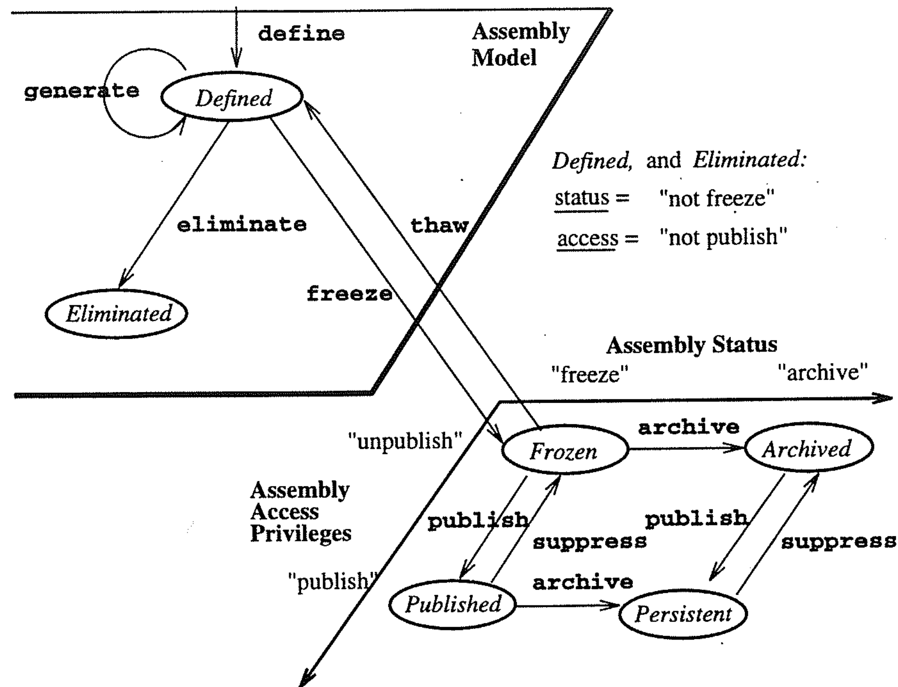



Figure 4.5: Assembly Model as a Finite State Machine

outside of the design team. The assembly has a status value, "archive," and an access value, "publish."

Figure 4.5 shows graphically the assembly model as a finite state machine. As mentioned in Section 2.3, *defined* and *eliminated* assemblies have a status property value, "not freeze," and an access property value, "not publish." The various assembly state operators are indicated in this figure by solid arrow lines. To ensure that a *published* assembly is not `eliminated` while it is being accessed, the model requires that an assembly must have a status value of "freeze," or "archive," before it can be `published`.

### 4.1.3 Relational Representation Scheme

We propose a scheme to represent assemblies that maintains both *composition* and *evolution* relationships among assemblies, and describes the property values assigned to each assembly in that discipline. This scheme stores assemblies in a discipline D as a relation, D-ASSEMBLY. A particular assembly is a tuple in this relation, and is uniquely identified by a system-generated attribute, *Assembly-id*. *Composition* relationships are maintained

by associating with an assembly definition the set of its component versions, *Components*. While *total* assemblies have a version from each entity in that discipline, a *partial* assembly contains versions from a subset of the entities. This scheme establishes *evolution* relationships by identifying the assembly from which a given assembly has been `generated`. Such parent links are stored in a *Parent-id* attribute. An assembly that is `defined` independently has no parent; the *Parent-id* attribute is assigned a "Null" value. Additionally, the assembly scheme identifies the status and access property values assigned to each assembly definition in *Fr-status* and *Pub-status* attributes. We now present the definition of a D-ASSEMBLY relation:

D-ASSEMBLY(*Assembly-id*, $Components_{i=1,n}$, *Assembly-type*, *Parent-id*, *Fr-Status*, *Pub-Status*)

where the attributes specified are:

- *Assembly-id:* System-generated attribute that uniquely identifies an assembly in a particular discipline.
- *Components:* Set of component versions of an assembly definition. The component set has $n$ elements, where $n$ is the total number of entities contained in the given discipline. A component version in a specific entity derivation hierarchy is identified by an attribute, *Version-id*. If the concerned assembly is a *partial* assembly, attributes corresponding to entities not included in the assembly definition are assigned "Null" values.
- *Assembly-type:* Indicator of a *total* assembly. Possible values are: *total* or *partial.*
- *Assembly-parent:* Identifier of the assembly from which the current assembly has been `generated`. For independently `defined` assemblies, the attribute is assigned a "Null" value.
- *Fr-Status:* Attribute that determines the status property value of an assembly. Possible values include "y" ("freeze"), "a" ("archive"), and "n" ("not freeze").
- *Pub-Status:* Attribute that determines the access property value of an assembly. Possible values include "y" ("publish"), and "n" ("not publish").

This scheme completely describes each assembly in a given discipline.

### 4.1.4 Extensions to the Version Scheme

The close coupling of assembly and version models implies that the status and access values assigned for a particular assembly definition are shared by each of its component versions. We enforce this restriction as preconditions on operators that assign specific assembly properties. The version model classifies versions into four additional version states based on the particular status and access property values. The four version states, *frozen*, *published*, *archived*, and *persistent* are similar to the state specifications proposed earlier for assemblies. Adding these four states to the ones proposed earlier, we can graphically represent the version model for collaboration as a finite state machine shown in Figure 4.6. As stated previously in Section 2.2, *active*, *suspended*, and *declared* versions have a status property value, "not freeze," and an access property value, "not publish." Status and access properties are not pertinent for *removed* versions. The various version state operators are indicated in this figure by solid arrow lines. `Freezing` a version definition ensures that it cannot be explicitly `removed`. This implies that the contents of a *frozen* version are also checkpointed and thus cannot be modified. Our model therefore requires that a version must be previously `declared`, before being assigned a status value, "freeze." Also, a *published* version must not be `removed` while designers from other disciplines are accessing it; only *frozen* or *archived* versions can be `published`.

We extend the representation scheme for versions that was presented earlier in Chapter 3 (Section 3.4.3) for locating access and status property values. We provide two new attributes, *Fr-status* and *Pub-status*, for the E-INDEX relation; the modified definition of the relation is:

E-INDEX(*Version-id*, *Parent-id*, *Decl-status*, *Fr-status*, *Pub-status*)

where the new attributes are:

- *Fr-status:* Maintains the status property of each version in an entity derivation hierarchy. Possible values include "y" ("freeze"), "a" ("archive"), and "n" ("not freeze").
- *Pub-Status:* Maintains the access property of each version in an entity derivation hierarchy. Possible values include "y" ("publish"), and "n" ("not publish").

In addition to the *external functions* given in Section 4.1.1 and Section 3.5.5, we outline the following functions that are used by operators to specify the version and assembly property values:

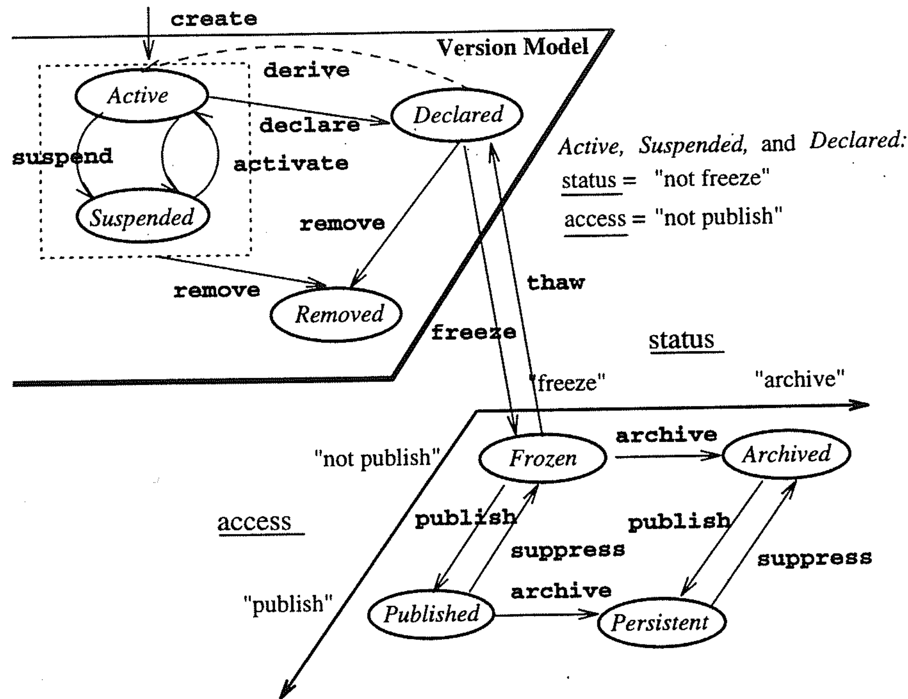



Figure 4.6: Version Model for Design Applications

- `mem-assembly` = (entity E, $id_{version}$ V) [1:n] $id_{assembly}$ A: Returns the set of assemblies in which the version V of entity E is a component.
- `status-assembly` = ($id_{assembly}$ A) string S: Obtains the status property value of an assembly A in the given discipline. Possible values of S are "freeze," "archive," and "not freeze."
- `access-assembly` = ($id_{assembly}$ A) string S: Obtains the access property value of an assembly A in the given discipline. Possible values of S are "publish," and "not publish."
- `comp-entity` = ($id_{assembly}$ A, integer I) entity E: Retrieves the name of entity E that is the $I_{th}$ component in the definition of assembly A.
- `comp-version` = ($id_{assembly}$ A, integer I) $id_{version}$ V: Retrieves the identifier of version V that is the $I_{th}$ element of an assembly A.

**Algorithm 19 (`freeze`)**

*Input: e:* Name of the concerned primitive entity.
 *v:* Identifier of the version being assigned the status value, "freeze."
*Output: success:* Boolean indicator of a successful operation.

```
procedure freeze = (entity e, id_version v) boolean success:
begin
   success := FALSE;
   if (vers-state(e, v) = declared) begin
      vers-state(e, v) := frozen;
   end;
   if ((vers-state(e, v) = frozen) OR (vers-state(e, v) = archived)) begin
      success := TRUE;
   end;
end procedure
◇
```

Figure 4.7: Procedure to `freeze` a Particular Version in an Entity Derivation Hierarchy

For the sake of clarity, we employ a construct "(ANY condition) that returns TRUE, if there exists a value for which the specified condition is satisfied. This construct is similar to the logical quantifier, $\exists$. For example, an expression, (ANY `access-assembly(mem-assembly(`*e*, *v*)) = "publish"), returns a TRUE value if the concerned version *v* of entity *e* is a component of even one assembly that has an access property value, "publish." The ANY construct can be easily implemented in ALGOL by checking the condition within a **while** loop.

We now discuss, in detail, the various version state operators to assign the status and access properties for a version definition.

- `freeze`(E, V): Specifies a *declared* version V of entity E as *frozen.* Successful execution of a `freeze` operation ensures that the version V is in the *frozen* or *archived* states. As the version model requires a version to be in the `declared` state before assigning a status value, "freeze," a `freeze` operation on an *active* or *suspended* version fails. An algorithm for a `freeze` operator is outlined in Figure 4.7.
- `thaw`(E, V): Specifies a *frozen* version V of entity E as *declared.* Figure 4.8 gives an algorithm for this operator. A `thaw` operation fails if either the version V does not initially have a status value, "freeze," or is a component of one or more *frozen* or *archived*

**Algorithm 20** (thaw)

*Input:* *e:* Name of the concerned primitive entity.
*v:* Identifier of the version being thawed.
*Output:* *success:* Boolean indicator of a successful operation.

```
procedure thaw = (entity e, id_version v) boolean success:
begin
  success := FALSE;
  if ((vers-state(e, v) = frozen) AND
      (NOT ((ANY status-assembly(mem-assembly(e, v)) = "freeze") AND
      (ANY status-assembly(mem-assembly(e, v)) = "archive")))) begin
    vers-state(e, v) := declared;
    success := TRUE;
  end;
end procedure
◇
```

Figure 4.8: Procedure to thaw a *Frozen* Version in an Entity Derivation Hierarchy

assemblies in that discipline. We implement these restrictions as a precondition for the thaw operator.

- publish(E, V): Specifies the <u>access</u> property value of version V of entity E as "publish." The version model requires <u>status</u> values, "freeze" or "archive" (*frozen* or *archived* states) for versions before they can be published. If successful, a *frozen* version is specified as *published*, while an *archived* version is now in the *persistent* state. We enforce this restriction as a precondition for the publish procedure (Figure 4.9).

- suppress(E, V): Specifies the <u>access</u> property value of a *published* or *persistent* version V as "not publish." The version V is now in the *frozen* or *archived* state. This operation can fail because the version: (i) has an initial <u>status</u> property value, "not freeze" (<u>access</u> value is also "not publish"), or (ii) is a component of at least one other *published* or *persistent* assembly in the same discipline D. We enforce these restrictions as preconditions for the proposed procedures in Figure 4.10.

- archive(E, V): Specifies the <u>status</u> value of a version V of entity E as "archive." If

**Algorithm 21** (`publish`)

*Input:* *e:* Name of the concerned primitive entity.
*v:* Identifier of the version being `published`.

*Output:* *success:* Boolean indicator of a successful operation.

**procedure** `publish` = (entity e, $\text{id}_{\text{version}}$ *v*) boolean *success*:
**begin**
  *success* := FALSE;
  **if** (`vers-state`(e, *v*) = *frozen*) **begin**
    `vers-state`(e, *v*) := *published*;
  **end**;
  **if** (`vers-state`(e, *v*) = *archived*) **begin**
    `vers-state`(e, *v*) := *published*;
  **end**;
**end procedure**
◇

Figure 4.9: Procedure to `publish` a Particular Version in an Entity Derivation Hierarchy

the version V was in the *frozen* state it is now *archived*, while a *published* version is now specified as *persistent.* Figure 4.11 shows an algorithm for the `archive` operator.

These version and assembly operators allow designers to share information with other disciplines, while retaining control over the actual design descriptions that they share.

### 4.1.5 Assembly State Operators

We now describe the operators to specify the various assembly states. These operators invoke corresponding version state operators to assign the required property values for each of its component versions that are not already in the specified state.

- `freeze-assembly`(D, A): Specifies a *defined* assembly A in discipline D as *frozen.* A successful execution of a `freeze-assembly` operation implies that each of the component versions of assembly A has a status value, "freeze" or "archive." The operator involves a `freeze` operator on those component versions which do not initially have the required status property value. Failing to assign a "freeze" status value for even

**Algorithm 22** (suppress)

*Input: e:* Name of the concerned primitive entity.
*v:* Identifier of the version being suppressed.
*Output: success:* Boolean indicator of the success of the operation.

```
procedure suppress = (entity e, id_version v) boolean success:
begin
  success := FALSE;
  if ((vers-state(e, v) = published) AND
      (NOT (ANY access-assembly(mem-assembly(e, v)) = "publish"))) begin
    vers-state(e, v) := frozen;
    success := TRUE;
  end;
  if ((vers-state(e, v) = persistent) AND
      (NOT (ANY access-assembly(mem-assembly(e, v)) = "publish"))) begin
    vers-state(e, v) := published;
    success := TRUE;
  end;
end procedure
◇
```

Figure 4.10: Procedure to suppress a *Published* or *Persistent* Version in an Entity Derivation Hierarchy

one of its component versions, causes the entire freeze-assembly operator to fail. Such a situation arises if the concerned component version is in the *active* or *suspended* states. Figure 4.12 outlines an algorithm to implement a freeze-assembly operator.

- thaw-assembly(D, A): Specifies a *frozen* assembly A in a particular discipline D as *defined.* Component versions of assembly D are not thawed, as they could be components of other assemblies having status values, "freeze" or "archive." An algorithm for this operator is shown in Figure 4.13.

- publish-assembly(D, A): Specifies the access property of an assembly A in discipline D as "publish." Since the contents of a *published* or *persistent* assembly is accessed by other designers, it must be guaranteed to exist at least while it is being referenced; an assembly has a status value, "freeze" or "archive," before it can be published. Thus,

**Algorithm 23** (archive)

*Input:* *e:* Name of the concerned primitive entity.
*v:* Identifier of the version being archived.

*Output:* *success:* Boolean indicator of a successful operation.

```
procedure archive = (entity e, id_version v) boolean success:
begin
   success := FALSE;
   if (vers-state(e, v) = frozen) begin
      vers-state(e, v) := archived;
      success := TRUE;
   end;
   if (vers-state(e, v) = published) begin
      vers-state(e, v) := persistent;
      success := TRUE;
   end;
   if (vers-state(e, v) = archive) begin
      success := TRUE;
   end;
end procedure
◇
```

Figure 4.11: Procedure to archive a Particular Version in an Entity Derivation Hierarchy

a *frozen* assembly is now in the *published* state, while an *archived* assembly is specified as *persistent*. The access value of an assembly is shared by its component version, a publish-assembly operator publishes component versions that do not have an initial access property value, "publish." Failure to publish even one component version causes the original publish-assembly to fail. Such situations occur when a component version does not have a status property value, "freeze" (the concerned version is *declared*, *active* or *suspended*). Figure 4.14 gives an algorithm for this operator.

- suppress-assembly(D, A): Specifies the access value of a *published* or *persistent* assembly A in a particular discipline D as "not publish." However, this operator does not explicitly suppress any of its component versions, as they could be components of other assemblies having an access property value, "publish." Figure 4.15 shows an

algorithm for the operator.

- `archive-assembly`(D, A): Specifies the status value of an assembly A in discipline D as "archive." Thus, *frozen* assembly is now *archived*, while a *published* assembly is specified as *persistent*. In addition, an `archive-assembly` operation shares the status value, "archive," with each component version. The original `archive-assembly` operator fails if any of its component versions cannot be `archived`. Such a situation occurs when a component version does not have a status value, "freeze;" it is originally in the *declared*, *active* or *suspended* states.

The current implementation of the assembly model does not **rollback** an assembly operation that fails. It does, however, maintain a log of versions whose states have been altered, giving designers information to manually undo any executed operators. This feature is based on our assumption that designers will often re-execute failed assembly operators after correcting the original causes of failure.

### 4.1.6 Assembly Change Management

We manage changes among assemblies along both the *composition* and *evolution* links. Along a *composition* axis, we describe an assembly as the result of a composite modeling operation (**union**) on its component versions. We implement a `display-assembly` operator that aggregates descriptions of its component versions to describe a complex entity. Along an *evolution* axis, we determine the differences between the descriptions of two assemblies, where one is an *ancestor* of the other. Such an *ancestor* relationship guarantees that each component version in the *ancestor* assembly is an *ancestor* of the corresponding component version (version of the same entity) in the *descendant* assembly. A `characterize-assembly-deltas` operator is implemented to report the changes between the two assemblies in terms of the changes between the descriptions of each of its component versions. The changes between each corresponding pair of versions is in turn determined using a `compute` operator.

**Algorithm 24** (`freeze-assembly`)

*Input: d:* Name of the concerned discipline.
*n:* Total number of entities in the given discipline.
*a:* Identifier of the assembly being assigned a status property value, "freeze."

*Output: success:* Boolean indicator of a successful operation.

```
procedure freeze-assembly = (discipline d, id_assembly a, integer n) boolean success:
begin entity e, id_version v, integer i;
  success := FALSE;
  if ((ably-state(d, a) = defined) OR
        (ably-state(d, a) = frozen)) begin
    success := TRUE;
    if (ably-state(d, a) = defined) begin
      for i = 1 to n begin
        if (success = TRUE) begin
          e := comp-entity(a, i);
          v := comp-version(a, i);
          success := freeze(e, v);
        end;
      end;
    end;
    if ((success = TRUE) AND (ably-state(d, a) = defined)) begin
      ably-state(d, a) := frozen;
    end;
  end;
end procedure
```
◇

Figure 4.12: Procedure to `freeze` a *Defined* Assembly in a Particular Discipline

**Algorithm 25** (`thaw-assembly`)

*Input:* *d:* Name of the concerned discipline.
  a: Identifier of the assembly being `thawed`.

*Output:* *success:* Boolean indicator of a successful operation.

**procedure** `thaw-assembly` = (discipline $d$, $\text{id}_{\text{assembly}}$ a) boolean *success*:
**begin**
  *success* := FALSE;
  **if** (`ably-state`($d$, a) = *frozen*) **begin**
    `ably-state`($d$, a) := *defined*;
  **end**;
**end procedure**
◇

Figure 4.13: Procedure to `thaw` a *Frozen* Assembly in a Particular Discipline

**Algorithm 26** (publish-assembly)

*Input:* $d$: Name of the concerned discipline.
$n$: Total number of entities in the given discipline.
a: Identifier of the assembly being published.

*Output:* *success:* Boolean indicator of a successful operation.

```
procedure publish-assembly = (discipline d, id_assembly a, integer n) boolean success:
begin entity e, id_version v, integer i;
   success := FALSE;
   if ((ably-state(d, a) = frozen) OR
         (ably-state(d, a) = archived) OR
         (ably-state(d, a) = published)) OR
         (ably-state(d, a) = persistent)) begin
      success := TRUE;
      if ((ably-state(d, a) = frozen) OR
            (ably-state(d, a) = archived)) begin
         for i = 1 to n begin
            if (success = TRUE) begin
               e := comp-entity(a, i);
               v := comp-version(a, i);
               success := publish(e, v);
            end;
         end;
      end;
      if ((success = TRUE) AND ((ably-state(d, a) = frozen) begin
         ably-state(d, a) := published;
      end;
      if ((success = TRUE) AND ((ably-state(d, a) = archived) begin
         ably-state(d, a) := persistent;
      end;
   end;
end procedure
◇
```

Figure 4.14: Procedure to publish a *Frozen* or *Archived* Assembly in a Particular Discipline

**Algorithm 27** (suppress-assembly)

*Input:* *d:* Name of the discipline in which the assembly is being created.
a: Identifier of the assembly in discipline *d* that is being suppressed.

```
procedure suppress-assembly = (discipline d, id_assembly a):
begin
   if ((ably-state(a) = published) begin
      ably-state(d, a) := frozen;
   end;
   if ((ably-state(a) = persistent) begin
      ably-state(d, a) := archived;
   end;
end procedure
◇
```

Figure 4.15: Procedure to suppress an Assembly with an Access Property, "publish"

**Algorithm 28** (`archive-assembly`)

*Input:* *d:* Name of the concerned discipline.
*n:* Total number of entities in the given discipline.
*a:* Identifier of the assembly being `archived`.

*Output:* *success:* Boolean indicator of the success of the operation.

```
procedure archive-assembly = (discipline d, id_assembly a, integer n) boolean success:
begin entity e, id_version v, integer i;
  success := FALSE;
  if ((ably-state(d, a) = frozen) OR
        (ably-state(d, a) = archived) OR
        (ably-state(d, a) = published)) OR
        (ably-state(d, a) = persistent)) begin
    success := TRUE;
    if ((ably-state(d, a) = frozen) OR
          (ably-state(d, a) = published)) begin
      for i = 1 to n begin
        if (success = TRUE) begin
          e := comp-entity(a, i);
          v := comp-version(a, i);
          success := archive(e, v);
        end;
      end;
    end;
    if ((success = TRUE) AND ((ably-state(d; a) = frozen) begin
      ably-state(d, a) := archived;
    end;
    if ((success = TRUE) AND ((ably-state(d, a) = published) begin
      ably-state(d, a) := persistent;
    end;
  end;
end procedure
```

◇

Figure 4.16: Procedure to `archive` a *Frozen* or *Published* Assembly in a Particular Discipline

## 4.2 Configuration Model

We propose a configuration as an integration framework to describe a multidisciplinary project design as an aggregation of design descriptions from each of the participating disciplines. Specifically, a configuration, in our model, is a set of *total* assemblies (one from each discipline) and an associated set of inter-disciplinary constraints. This section is divided into four parts. We first provide an overview of the configuration model. In this model, configurations are created by an individual designer using a design from his/her own discipline along with designs from each of the remaining participating disciplines. We provide access and status properties for configuration definitions for promoting cooperation among project team members. Because of the close coupling of the version, assembly, and configuration models, these configuration properties translate into requirements on component assemblies (and in turn their own component versions). Third, we present a scheme to represent the configuration model in a relational environment, establishing both *composition* and *evolution* links among configurations. This scheme also identifies the status and access property values of each configuration created by that discipline. Finally, the last part of this section discusses the change management capabilities of our model; we characterize changes among configurations along both the *composition* and *evolution* links. The changes are expressed in terms of changes among component versions and assemblies, and are essential to both project monitoring and coordination.

### 4.2.1 Overview of the Configuration Model

Based on its existence, we classify a configuration definition in one of the following two states:

- *Defined configuration,* which describes a project design in terms of component *total* assemblies, one from each of the participating disciplines, and an associated set of inter-disciplinary project constraints.
- *Eliminated configuration,* which was previously *defined* but no longer exists.

A configuration can be either `defined` independently, or `generated` as a child of an existing *defined* configuration. In either case, the newly created configuration is in the *defined* state. A `generate` operation substitutes one or more components of the parent configuration with more refined designs from those disciplines, thereby tracking an evolving multidisciplinary

project description. Each component assembly in the parent configuration is guaranteed to be an *ancestor* of the corresponding component assembly (design description from the same discipline) in the child configuration. A transitive closure of parent links between configuration definitions establishes *ancestor-descendant* relationships among them. Also, an `eliminate` operation removes an existing *defined* configuration to reduce the required storage space.

A designer creates (`defines` or `generates`) a configuration, using a design from his/her own discipline along with a design description from each of the remaining participating disciplines. To provide a stable environment for sharing information, we place the following two restrictions on component designs of a configuration definition:

1. Individual designs and their components must not be `eliminated` while they are included in a configuration definition. In other words, a *total assembly* must be guaranteed to exist, to be a component of a configuration.

2. Descriptions of designs from the other disciplines must be accessible to a designer in order to include them in his/her configuration.

Therefore to satisfy these requirements for a *defined* configuration, we establish that component *total* assemblies from disciplines other than the creator's discipline must be *published*, while the component assembly from the creator's own discipline need only be *frozen*.

Before describing the configuration state operators in detail, we outline the data type declarations employed in the discussions. These data type declarations are in addition to those provided in Sections 4.1.1.

- $\mathrm{id}_{\mathrm{config}}$: Type declaration of the identifier of the concerned configuration definition created in a particular discipline.

- $\mathrm{state}_{\mathrm{config}}$: State specification of a configuration definition.

We also use the following *external functions* to develop algorithms for the proposed assembly state operators:

- `config-numgen` = (discipline D, [1:m] $\mathrm{id}_{\mathrm{assembly}}$ A, $\mathrm{id}_{\mathrm{config}}$ R) $\mathrm{id}_{\mathrm{config}}$ F: Generates an identifier F for a new configuration created by a designer from a discipline D that is `generated` as a child of an existing *defined* configuration R, and has a component *total* assembly from each of the $m$ participating disciplines. When independently `defining`

a new configuration, the parent identifier $R$ is assigned a value, "Null." This function also stores the configuration definition in terms of its component *total* assemblies, which can be retrieved by other operators.

- `config-parent` = (discipline D, $\text{id}_{\text{config}}$ F) $\text{id}_{\text{config}}$ R: Returns the identifier of configuration R that is the parent of configuration F belonging to the same discipline D. The function returns the identifier of the configuration from which the current configuration has been `generated`. If the configuration F were `defined` independently, this function returns a "Null" value.

- `config-state` = (discipline D, $\text{id}_{\text{config}}$ F) $\text{state}_{\text{config}}$ X: This function obtains the current state specification X of a configuration F created by a designer from a discipline D.

- `config-destroy` = (discipline D, $\text{id}_{\text{config}}$ F) boolean Z: Erases a configuration definition F from the set of configurations belonging to discipline D. Z indicates a successful execution of the operator.

- `comp-config`(discipline D, $\text{id}_{\text{config}}$ F, discipline E) $\text{id}_{\text{assembly}}$ A: Returns the *total* assembly in participating discipline E that is a component of configuration F created by a designer from discipline D.

For the sake of clarity, we employ a construct, "(ALL condition)," which returns a TRUE value if the condition is satisfied for all the considered values. For example, the expression, (ALL ((`access-assembly`(`comp-config`($d$, $c$, $e$)) = "publish") AND ($e \neq d$))), is TRUE if each component assembly of configuration $f$, which is not from the discipline $d$ that created the configuration, has an access property value "publish." This construct is similar to the logical quantifier, $\forall$. We can easily implement the above expression in ALGOL by specifying the condition within a **for** loop.

We provide the following configuration state operators:

- `define-config`(D, [1:m] A) F: Creates a new configuration in a discipline D, given a set of component *total* assemblies, one from each of the $m$ participating disciplines. A `define-config` operator invokes a `config-numgen` function which generates the system identifier F for the newly `defined` configuration. Figure 4.17 gives an algorithm to implement a `define-config` operation.

**Algorithm 29** (`define-config`)

*Input:* *d:* Name of the discipline that is `defining` the configuration.
*a:* Set of component *total* assemblies, one from each of the $m$ participating disciplines.
*Output:* *f:* Identifier of the new configuration, which is in the *defined* state.

**procedure** `define-config` = (discipline $d$, [1:m] $\text{id}_{\text{assembly}}$ a) $\text{id}_{\text{config}}$ *f*:
**begin**
  **if** (ALL ((`access-assembly`(`comp-config`($d$, $c$, e)) =
      "publish") AND ($e \neq d$))) **begin**
    $f$ := `config-numgen`($d$, [1:m] a, "Null");
    `config-state`($d$, $f$) := *defined*;
  **end**;
**end procedure**
◇

Figure 4.17: Procedure to `define` a New Configuration

- `generate-config`(discipline D, [1:m] $\text{id}_{\text{assembly}}$ A, $\text{id}_{\text{config}}$ R) $\text{id}_{\text{config}}$ F: Creates a new configuration as a *descendant* of a given configuration R belonging to the same discipline D. A `generate-config` operator uses a `config-numgen` function for obtaining a unique system identifier F for the new configuration. This operator ensures that each component assembly of the new configuration F is a *descendant* of the corresponding assembly (assembly from the same discipline) in configuration R. A procedure to `generate` a new configuration is given in Figure 4.18.

- `eliminate-config`(discipline D, $\text{id}_{\text{config}}$ F): Specifies an existing *defined* configuration F belonging to a discipline D to be in the *eliminated* state. However, this operator does not further `eliminate` any of its component assemblies which can be components of other configurations. Figure 4.19 outlines an algorithm to execute this operator.

These three operators specify the two assembly states, *defined* and *eliminated.*

### 4.2.2 Configuration Properties for Collaboration

Given that configurations are frameworks to represent multidisciplinary project descriptions, we can provide access and status properties for configuration definitions to simulate real situations found in typical collaborative environments. Based on the specific property values

**Algorithm 30** (`generate-config`)

*Input:* *d:* Name of the discipline which is `generating` the new configuration.

*r:* Identifier of a *defined* configuration that is an *ancestor* of the newly created configuration.

*a:* Identifiers of *m* component *total* assemblies of the configuration being created. Each component assembly is a *descendant* of the corresponding assembly (assembly from the same discipline) in configuration *r*.

*Output:* *f:* Identifier of the newly `generated` configuration which is in the *defined* state.

**procedure** `generate-config` = (discipline *d*, [1:m] $\text{id}_{\text{assembly}}$ a, $\text{id}_{\text{config}}$ r) $\text{id}_{\text{config}}$ *f*:
**begin**
  **if** (ALL ((`access-assembly`(`comp-config`(*d*, *c*, *e*)) =
      "publish") AND (e $\neq$ *d*))) **begin**
    *f* := `config-numgen`(*d*, [1:m] $\text{id}_{\text{assembly}}$ a, $\text{id}_{\text{config}}$ r);
    `config-state`(*d*, *f*) := *defined*;
  **end**;
**end procedure**
◇

Figure 4.18: Procedure to `generate` a New Configuration from an Existing Configuration Definition

assigned to it, a configuration definition can be categorized into one of the following four states:

- *Intermediate configuration,* which allows a designer to privately evaluate one or more solution alternatives with respect to designs made accessible by other designers. Feedback from such evaluations help designers independently refine their individual designs for efficient integration with the overall project. An *intermediate* configuration cannot be accessed by other disciplines; access value is "not publish." In addition, such a configuration cannot be explicitly `eliminated`; the status property is assigned a value, "freeze." To satisfy the configuration properties, the component assemblies from other disciplines must at least be in the *published* state, whereas the component assembly from the designer's own discipline need only be *frozen.*

- *Accessible configuration,* which simulates a meeting scenario in which each designer brings his/her design to the table for the entire team to collectively evaluate the

**Algorithm 31** (`eliminate-config`)

*Input:* *d:* Name of the concerned discipline.
*f:* Identifier of the *defined* configuration that is being `eliminated`.
*Output:* *success:* Indicator of the successful execution of the `eliminate` operation.

```
procedure eliminate-config = (discipline d, id_config f) boolean success:
begin
  success := FALSE;
  if (config-state(f) = defined) begin
    config-state(f) := eliminated;
    success := config-destroy(d, f);
  end;
end procedure
◇
```

Figure 4.19: Procedure to `eliminate` a *Defined* Configuration

entire project description, and to identify any inconsistencies among its individual components. Such a project design can be referenced by the entire design team; the configuration has an access property value, "publish." Since the configuration can be referenced by other disciplines, it must also be guaranteed to exist (status value, "freeze"). Thus, each component assembly of an *accessible* configuration must be in the *published* or *persistent* state.

- *Landmark configuration,* which represents a consistent project description that can be shared with actors outside the design team. Such configurations represent, among others: (i) team records checkpointing descriptions of the project at the end of specific design phases, (ii) project designs submitted to regulatory agencies for construction approval, and (iii) documents released to contractors for bidding purposes. *Landmark* configurations are typically maintained for extended periods, often covering the facility's entire life cycle. Thus, such designs have a status value, "archive." Also, such configurations can be referenced by the entire team (access value, "publish"). To satisfy these properties, component assemblies of a *landmark* configuration must have a status property value, "archive," as well as an access value, "publish." Therefore each component assembly must be in the *persistent* state.

Table 4.1: Specifications of States of Component Assemblies

| Configuration State | Assembly State (Other Disciplines) | Assembly State (Owner Discipline) |
|---|---|---|
| *Intermediate* | *Published* | *Frozen* |
| *Accessible* | *Published* | *Published* |
| *Recorded* | *Persistent* | *Archived* |
| *Landmark* | *Persistent* | *Persistent* |

- *Recorded configuration,* which represents a project description containing a component design alternative not selected for the current project, but maintained for future reference. This would allow a mechanical engineer, for example, to maintain, as personal records, an alternative ducting layout that was not selected for the concerned project. A *recorded* configuration has a status property value, "archive," but remains inaccessible to designers from the other disciplines. To satisfy these semantics, the assembly included from the creator's discipline need only be *archived*, while assemblies from the other disciplines must be *persistent.*

Figure 4.20 graphically represents the configuration model as a finite state machine. As established in Section 2.4.2, *defined* and *eliminated* configurations have a status property value, "not freeze," and an access property value, "not publish." In this figure, we identify the configuration states and provide operators to specify them. Table 4.1 summarizes the inclusion rules for component assemblies of the various configuration states. These state specifications represent the minimal requirements of status and access properties needed for assemblies to be components of a configuration specified to be in a given state. The close coupling of the assembly and version models necessarily enforces these rules on versions *included* in the component assemblies.

We now describe the configuration state operators in greater detail.

- `protect`(D, F): A `protect` operation assigns a status value, "freeze," for a configuration F belonging to the discipline D. Such a configuration cannot be `eliminated` by a designer from the same discipline unless explicitly `unprotected`. Since the minimal property specifications for component assemblies of an *intermediate* configuration are the same as those for a *defined* configuration, this operator does not concern with the component assembly states. Figure 4.21 outlines a procedure to implement the

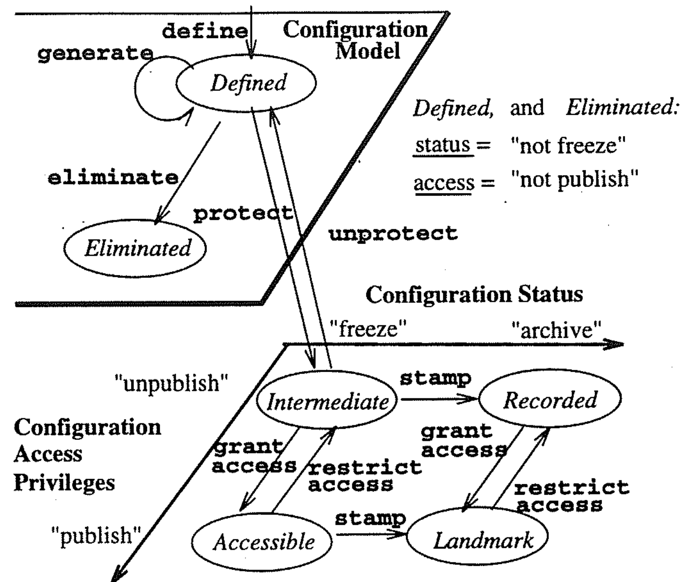



Figure 4.20: Configuration Model to Support Collaboration

`protect` operator.

- `unprotect`(D, F): An `unprotect` operation initializes the <u>status</u> property value of a configuration F from discipline D from "freeze" to "not freeze"; an *intermediate* configuration can now be `eliminated`. In the context of the configuration model, the operator specifies an *intermediate* configuration F in the discipline D as *defined.* An algorithm for this operator is presented in Figure 4.22.

- `grant-access`(D, F): A `grant-access` operator sets an <u>access</u> property value, "publish," for a configuration F from a discipline D. The definition as well as contents of the configuration are now accessible to designers from other disciplines. Therefore the configuration model ensures that the concerned configuration be in the *intermediate* or *recorded* states before it can be `published`. Furthermore, its component assemblies also must have an <u>access</u> value, "publish." Since component assemblies from other disciplines are guaranteed to have an <u>access</u> value, "publish," the operator needs to only check if the assembly from the designer's own discipline D has an <u>access</u> property value, "publish." We enforce the above two requirements as preconditions on the operator. Figure 4.23 gives an algorithm for this operator. Particular to the configuration model, a `grant-access` operator specifies an *intermediate* configuration F

**Algorithm 32** (`protect`)

*Input: d:* Name of the concerned discipline.

*f:* Identifier of the configuration belonging to discipline *d* that is being `protected`.

*Output: success:* Boolean indicator of a successful operation.

**procedure** `protect` = (discipline *d*, $\text{id}_{\text{config}}$ *f*) boolean *success*:
**begin**
*success* := FALSE;
**if** (`config-state`(*d*, *f*) = *defined*) **begin**
`config-state`(*d*, *f*) := *intermediate*;
*success* := TRUE;
**end**;
**end procedure**
◇

Figure 4.21: Procedure to `protect` a *Defined* Configuration

belonging to discipline D as *accessible*. Alternatively, a configuration in the *recorded* state is now specified as *landmark*.

- `restrict-access`(D, F): A `restrict-access` operation revokes access privileges granted to designers from other disciplines on a particular configuration F created by a designer from a discipline D. The operation sets the access property value of the configuration as "not publish," while the property values of its components remains unchanged. In other words, the operator specifies an *accessible* or *landmark* configuration F from a discipline D as *intermediate* or *recorded*, respectively. An algorithm for this operator is shown in Figure 4.24.

- `stamp`(D, F): A `stamp` operation sets the status property value of a configuration F from discipline D as "archive," guaranteeing its existence for the life cycle of the facility. Before we can specify an "archive" status value, the configuration must have a status value, "freeze" (*intermediate* or *accessible* states). Also, each component assembly must have a status value, "archive." The required access property values for the component assemblies are ensured by the initial states of the configuration F; assemblies from other disciplines already have an access value, "publish," while if the configuration F is *intermediate* the designer's own component assembly could be

**Algorithm 33** (`unprotect`)

*Input: d:* Name of the concerned discipline.
*f:* Identifier of the configuration of discipline $d$ that is being `unprotected`.
*Output: success:* Boolean indicator of a successful operation.

**procedure** `unprotect` = (discipline $d$, $\text{id}_{\text{config}}$ $f$) boolean *success*:
**begin**
  *success* := FALSE;
  **if** (`config-state`($d$, $f$) = *intermediate*) **begin**
    `config-state`($d$, $f$) := *defined*;
    *success* := TRUE;
  **end**;
**end procedure**
◇

Figure 4.22: Procedure to `unprotect` a *Defined* Configuration

assigned an access property value, "not publish". A successful operation specifies an *intermediate* configuration as *recorded*, while an *accessible* configuration is now in the *landmark* state. An algorithm for the `stamp` operator is shown in Figure 4.25.

The close coupling of the three layers of our model requires that the properties of component *total* assemblies (and their component versions) not be changed while the assemblies are still component of existing configurations. We enforce these requirements as preconditions on operators that specify access and status properties for versions and assemblies that are included in an existing configuration definition. For example, a *published* assembly or version cannot be `suppressed` while it is a component of any *accessible* or *landmark* configuration (having an access value, "publish") created by that discipline, or is a component of any configuration definition created by other disciplines. Similarly, a *frozen* assembly or any of its *included* versions cannot be `thawed` while being a component of any existing configuration.

**Algorithm 34** (`grant-access`)

*Input:* $d$: Name of the concerned discipline.
$f$: Identifier of the configuration of discipline $d$ that is being `granted-access`.
*Output:* *success:* Boolean indicator of a successful operation.

**procedure** `grant-access` = (discipline $d$, $\text{id}_{\text{config}}$ $f$) boolean *success*:
**begin**
*success* := FALSE;
**if** ((`config-state`($d$, $f$) = *intermediate*) AND
(`access-assembly`(`comp-config`($d$, $f$, $d$)) = "publish")) **begin**
`config-state`($d$, $c$) := *accessible*;
**end**;
**if** ((`config-state`($d$, $f$) = *recorded*) AND
(`access-assembly`(`comp-config`($d$, $f$, $d$)) = "publish")) **begin**
`config-state`($d$, $c$) := *landmark*;
**end**;
**end procedure**
◇

Figure 4.23: Procedure to `grant-access` to an *Intermediate* or *Recorded* Configuration

### 4.2.3 Relational Representation Scheme

We propose a scheme to represent configurations in a relational environment. This scheme maintains configurations created by a discipline D as a relation D-CONFIGURATION, establishing both the *composition* and *evolution* relationships among configurations. The scheme also stores the <u>status</u> and <u>access</u> property values assigned to each configuration definition. We establish the *composition* relationships by identifying all component *total* assemblies of each configuration; a *Components* set, in this scheme, contains all the component assembly identifiers. On the other hand, we establish *evolution* relationships by identifying the parent configuration from which a given configuration has been `generated`. Such parent links are stored in a *Parent-id* attribute. For configurations that have been `defined` independently, this attribute is assigned a value, "Null."

The D-CONFIGURATION relation has the following definition:

D-CONFIGURATION(<u>*Config-id*</u>, $\text{Components}_{\{i|i=1,\ldots,m\}}$ , *Config-parent*, *Fr-status*, *Pub-status*)

**Algorithm 35** (`restrict-access`)

*Input:* *d:* Name of the concerned discipline.

*f:* Identifier of the configuration belonging to the discipline *d* whose access privileges are being `restricted`.

*Output:* *success:* Boolean indicator of a successful operation.

**procedure** `restrict-access` = (discipline *d*, $\text{id}_{\text{config}}$ *f*) boolean *success*:
**begin**
  *success* := FALSE;
  **if** (`config-state`(*d*, *f*) = *accessible*) **begin**
    `config-state`(*d*, *f*) := *intermediate*;
  **end**;
  **if** (`config-state`(*d*, *f*) = *landmark*) **begin**
    `config-state`(*d*, *f*) := *recorded*;
  **end**;
**end procedure**
◇

Figure 4.24: Procedure to `restrict-access` to an *Accessible* or *Landmark* Configuration

where the attributes specified are:

- *Config-id:* System-generated attribute that uniquely identifies each configuration created by a given discipline.
- *Components:* Set of *m total* assemblies that compose a configuration definition, one assembly from each participating discipline.
- *Config-parent:* Identifier of the parent configuration from which the current configuration has been `generated`. If `defined` independently, this attribute is assigned a "Null" value.
- *Fr-Status:* Specifies the status property value of a configuration. Possible values include "y" ("freeze"), "a" ("archive"), and "n" ("not freeze").
- *Pub-Status:* Specifies the access property value of a configuration. Possible values include "y" ("publish"), and "n" ("not publish").

The proposed representation scheme completely describes the *composition* and *evolution*

**Algorithm 36** (stamp)

*Input:* *d:* Name of the concerned discipline.
*f:* Identifier of the configuration belonging to the discipline *d* that is being `stamped`.
*Output:* *success:* Boolean indicator of a successful operation.

**procedure** `stamp` = (discipline *d*, $\text{id}_{\text{config}}$ *f*) boolean *success*:
**begin**
*success* := FALSE;
**if** ((`config-state`(*d*, *f*) = *intermediate*) AND
(ALL (`status-assembly`(`comp-config`(*d*, *f*, e)) = "archive"))) **begin**
`config-state`(*d*, *f*) := *recorded*;
**end**;
**if** ((`config-state`(*d*, *f*) = *accessible*) AND
(ALL (`status-assembly`(`comp-config`(*d*, *f*, e)) = "archive"))) **begin**
`config-state`(*d*, *f*) := *landmark*;
**end**;
**end procedure**
◇

Figure 4.25: Procedure to `stamp` an *Intermediate* or *Accessible* Configuration in a Particular Discipline

relationships among configurations belonging to a given discipline, and maintains their status and access values.

### 4.2.4 Configuration Change Management

We manage changes among configurations along both the *composition* and *evolution* relationships. Along the *composition* axes, we describe an overall project design in terms of individual designs from the participating disciplines. We implement a `display-configuration` operator that invokes a `display-assembly` operator to describe each component design. Along the *evolution* axes, we determine the differences between two configurations, where one is an *ancestor* of another. Such an *ancestor* relationship guarantees that each component *total* assembly in the ancestor configuration is also an *ancestor* of the corresponding component assembly (assembly from the same discipline) in the second (*descendant*) configuration. We implement a `characterize-config-deltas` operator that represents the differences between two concerned configurations as the aggregation of the differences between each pair

of corresponding component assemblies; differences between each component assembly pair are in turn described by executing `characterize-assembly-deltas` operations.

## 4.3 Summary and Discussions

In this chapter, we have described the assembly and configuration models and proposed a scheme to represent them in a relational environment. Configurations provide a framework that integrates designs from the participating disciplines to describe an overall project. We have identified the basic status and access properties for configurations to support cooperation among designers. Based on its particular properties, we classify a configuration into one of the four states. Using an *intermediate* configuration, a designer can privately evaluate his/her design with respect to the entire project. *Accessible* configurations simulate meeting scenarios in which each designer brings the design to the table, allowing the entire team to collectively evaluate the design progress. *Landmark* configurations represent project descriptions that are maintained for extended periods and are accessible to members both within and outside the design team. These include (i) team records checkpointing project descriptions at the end of specific design phases, (ii) project designs submitted to regulatory agencies for construction approval, and (iii) documents released to contractors for bidding purposes. Finally, *recorded* configurations allows designers to maintain, for personal references, alternative designs that were not selected for the current project.

Assemblies, in our model, aggregate component versions to describe complex entities. Assemblies can be either *total* or *partial*. A *total* assembly represents a complete design in an individual discipline, and contains a version of each entity belonging to that discipline. *Partial* assemblies, on the other hand, describe complex entities that could be further aggregated to describe more complex entities or a complete design in that discipline.

A salient feature of our model is the close coupling of the version, assembly, and configuration layers. To provide a stable environment for sharing information, we specify restrictions on *total* assemblies that are components of the various configuration states. To satisfy these restrictions, we provide access and status properties for assemblies, classifying them into four distinct states based on the property assignments. Furthermore, the close coupling of the assembly and version models ensures that the status and access property of an assembly definition is in turn shared by each of its component versions.

Also, in this chapter, we propose schemes to represent both the assembly and configuration models in a relational environment. These schemes establish both the *composition* and *evolution* relationships. While *composition* links identify the components of a configuration or assembly. an *evolution* link identifies the parent from which the concerned assembly or

configuration has been `generated`. The close coupling of the three layers allows `computed` version changes to be combined to characterize changes at the assembly and configuration levels, supporting both coordination and monitoring activities.

Many proposed versioning schemes have incorporated a notion of version states to support collaboration [20, 8, 34]. Chou and Kim [8] have enumerated a number of interesting version states (*transient*, *working*, and *released*), and have proposed an architecture of databases (*public*, *group*, and *private*) to store shared versions. Unfortunately, this scheme explicitly associates the properties of a given version with the database in which it is located. This results in two major drawbacks. First, it is typically hard to realize the proposed database architecture in real design scenarios. Second, to modify the accessibility of a design version, we need to explicitly copy the version from one database to another. The Electronic Document Management System (EDMS) [34] project also specifies a state graph for releasing a document version. The system provides four states, *draft*, *ready*, *checked*, and *approved*, which correspond to approval status of a given document version. Interestingly, they provide *authorizing* and *commit* programs to specify how versions should behave in different situations. Although this mechanism is more flexible than schemes with prespecified state transition rules, the choice of state definitions is itself quite arbitrary. None of the surveyed versioning schemes have systematically identified the core version properties necessary for collaboration.

Some research efforts have previously defined configurations as the version of a composite entity in terms of the versions of its components [20, 23, 32, 28]. Strategies, such as layers and contexts [20], and version environments [32] have been proposed to to select versions that are included in a consistent configuration. Cellary et al. [7] uses a concept of database versions to control the propagation of versions of composite entities as a response to new versions of it component entities. However, none of the surveyed works support configurations for multidisciplinary projects in distributed paradigms.

In conclusion, we believe that our three-layered model provides a comprehensive solution for project change management.

# Chapter 5

# Change Management in a CAD Environment

This chapter presents a scheme for representing versions, assemblies, and configurations in a CAD environment and provides operators for storing and managing changes at each of the three levels. In the CAD context, an instance of a primitive entity is represented as a *list* composed of associated lists, where each associated list consists of an attribute *name* and *value* pair. Attributes can be either *primary* or *secondary*. The set of primary attribute values uniquely identify an instance that is contained in a particular version of an entity. Secondary attributes represent additional design properties of the instance. A change is a data operation on an instance of an entity. CAD systems provide several operators to manipulate design instances. We, however, abstract available CAD drawing and editing operators into three primitive operators that capture their essence: `insert`, `delete` and `replace`. For example, a CAD operation that modifies the geometry (**scale**) or spatial arrangements (**move** or **rotate**) of an instance are mapped to a `replace` primitive operation. **Drawing** a new instance is an `insert` primitive operation, while **erasing** an existing instance corresponds to a `delete` primitive operation. We thus model a change as a primitive `insert`, `delete`, or `replace` operation on an instance.

We conceptually extend the relational scheme presented in Chapter 4 for representing assemblies. An assembly, in our model, represents a complex entity resulting from a composite modeling operation on a set of component instances. Our implementation currently handles three composite modeling operations: **union**, **intersect** and **subtract**, which are also

supported by AUTOCAD's Advanced Modeling Extension (AME). An individual component of an assembly can be an instance of either a primitive or complex entity. While the former is an instance belonging to a specific version in the entity derivation hierarchy, the latter is represented by another assembly. We develop a *component hierarchy* for a given assembly by recursively expanding its definition. The root of this hierarchy is the original assembly; the leaf nodes represent primitive instances *included* in the assembly definition. Formally, an instance of a primitive entity is *included* in an assembly if it is either a component of that assembly definition or is *included* in one of its components. Furthermore, versions containing instances that are *included* in a given assembly are denoted as *included* versions. The model determines the changes between versions; we characterize changes at the assembly and configuration levels by recursively aggregating computed version changes along both their *composition* and *evolution* relationships.

We have implemented the proposed data management model as a prototype in an AUTOCAD system Release 12. AUTOLISP provides the programming interface. We illustrate different aspects of our model using an integrated facility design example. The described example is simple yet realistic, and has been tested on the CAD prototype system. Design entities in the prototype are represented as 3-D graphical objects which have been defined using AUTOCAD's AME solid modeler.

The rest of this chapter is organized into four parts. We first describe the Medical Cyclotron facility example which is used throughout this chapter to illustrate the various aspects of the model. This example was first introduced in Chapter 2, and is developed here in greater detail. Second, we propose a CAD representation scheme for the version model. A version, in this scheme, contains the set of *equivalent primitive operations* on each instance that was modified when that version was in the *active* state. We employ a *check-out/check-back-deltas* protocol to structure the interaction between an application session and the *active* version and describe CAD operators for detecting, storing, and managing changes among versions represented in this environment. Third, we develop a recursive scheme to represent the assembly model in a CAD paradigm. Management of changes along both *composition* and *evolution* relationships is also discussed. Finally, we describe a scheme to represent configurations, and demonstrate the model's capabilities for project **coordination** and **monitoring**.

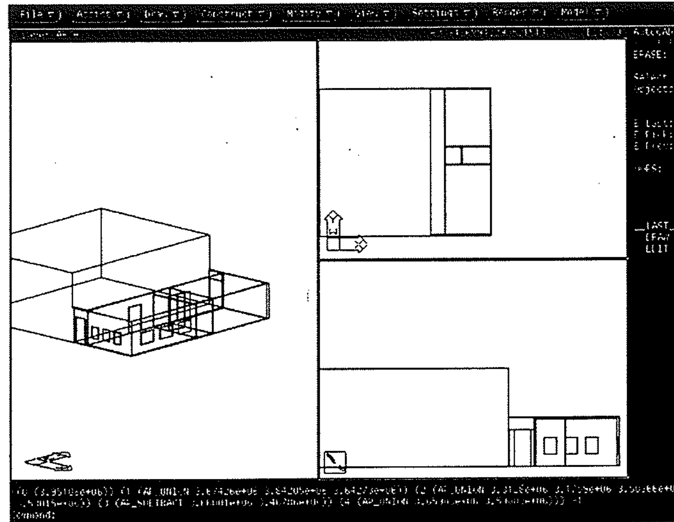


Figure 5.1: Initial Floor Plan of the Facility (Assembly aa-i0)

## 5.1 Application Example

Figure 5.1 shows an initial architectural layout of a Medical Cyclotron facility. To keep the example simple, we focus on only three of the many participating disciplines: architecture, structural engineering and mechanical engineering. The facility is stored in the AUTOCAD environment using AME (AUTOCAD's Modeling Extension) 3-D BOX objects.

## 5.2 Version Model

We present a CAD implementation scheme for the version model and describe how operators manage changes among individual versions for this given scheme.

### 5.2.1 Representation Scheme

We present a *forward deltas* scheme [36] to implement the version model in a CAD environment. Similar to the relational representation scheme given in Section 3.4, in a CAD environment, we completely describe a primitive entity E using three lists: E-DATA, E-INDEX, and E-ACTIVE. An E-DATA list associates the description of each version with a system-generated identifier, *Version-id*. Each version, in this scheme, contains a summary

of all changes made on that entity while the concerned version was *active*. Instances are described by a list of attribute name and value pairs; a change, in our model, is an `insert`, `delete` or `replace` primitive operation on an instance description. An attribute in the E-DATA relation, *Op-desc*, records the net equivalent operator on each instance contained in the considered version. For redesign, we also specify an *Anc* attribute that links each instance with its most recent description in an *ancestor* version. For computational efficiency, this "optimized" version scheme explicitly maintains intermediate *complete* versions. An E-INDEX list implicitly maintains an entity derivation hierarchy, and identifies *declared* versions. A *Parent-id* attribute stores the identifier of the parent of each version. As the version set is a tree structure, all versions except the root have exactly one parent version; a version tree can therefore be uniquely generated with this information. A *Version-type* attribute in the E-INDEX list distinguishes those versions which are *complete*. Finally, an E-ACTIVE list explicitly locates the currently *active* version. A version that is neither *declared* nor *active* is inferred as *suspended*.

We now present the definitions of the three lists,

- E-INDEX: ( (<u>*Version-id*</u> Value)
  ( (*Version-type* Value)
  (*Parent-id* Value)
  (*Decl-Status* Value))) $_{\{i \mid i = 1,\ldots, v\}}$

  which has *v* versions in the entity hierarchy. The remaining attributes are given as follows:

  - *Version-id:* System-generated identifier of the version in an entity derivation hierarchy.
  - *Version-type:* Indicator of *complete* versions. The root of an entity derivation hierarchy is always *complete*.
  - *Parent-id:* Identifier of the parent of a given version. The root version of an entity derivation hierarchy has a "Null" value.
  - *Decl-status:* Indicator of *declared* versions, which has the possible values: "y" or "n".

- E-ACTIVE: (<u>*Act-Version-id*</u> Value)

where the attribute *Act-version-id* identifies the currently *active* version.

- E-DATA: ( (<u>*Version-id*</u> Value)

  $( (\textit{Primary-attribute}\ \text{Value})_{\{i \mid i = 1,\ldots, p\}}$
  $(\textit{Secondary-attribute}\ \text{Value})_{\{j \mid j = 1,\ldots, s\}}$
  $(\textit{Operator-descriptor}\ \text{Value})$
  $(\textit{Anc}\ \text{Value}))_{\{k \mid k= 1,\ldots, n\}}\ )_{\{l \mid l = 1,\ldots, v\}}$

  which maintains $v$ versions of entity E; a version in this scheme is describe by $n$ changes. The remaining attributes are specified as follows:

  - *Version-id:* Identifier of the version in an entity derivation hierarchy.
  - *Primary-Attribute:* Set of $p$ primary attributes in the entity scheme that uniquely identify an instance in the given version.
  - *Secondary-Attribute:* Set of $s$ secondary attributes in the entity scheme that describe certain design properties of each instance. Such attributes are functionally dependent on the set of primary attributes.
  - *Operator-descriptor:* Operator that summarizes all changes made to the concerned instance when the given version was *active.*
  - *Anc:* Identifier of the ancestor version in which the given instance was most recently modified. In situations where the instance was `inserted` in the particular version, this attribute is assigned a "Null" value. Although an *Anc* value can be alternatively computed by retracing the derivation path from a version through its *ancestors*, it is critical in redesign situations to quickly identify earlier descriptions from which to redesign. We therefore explicitly store this value to reduce the search time.

These three lists completely describe a primitive entity.

Figure 5.2 shows an example derivation hierarchy of a BOX primitive entity. The BOX instances in each version of the hierarchy represent the exterior architectural walls of the "running" facility design shown in Figure 5.1. In this figure, versions b-1, b-1a0, and b-1a1 are *declared*; version b-1a2 is *suspended*, while version b-2 is *active.* Specific wall instances in a given version are represented by the provided scheme and are uniquely identified by the value assigned to its primary attribute, *Box-id*; the attribute *Box-id* (generated using

Table 5.1: Example Sequence of Version Operations on the BOX Entity

| Operations | Version Affected | Initial State | Final State |
|---|---|---|---|
| `activate`(BOX, b-1a2) | b-2<br>b-1a2 | *active*<br>*suspended* | *suspended*<br>*active* |
| `declare`(BOX, b-1a2) | b-1a2 | *active* | *declared* |
| `derive`(BOX, b-1a2) | b-1a3 | non-existent | *active* & child of version b-1a2 |

AUTOCAD's handle descriptor "5") uniquely identifies an instance of a 3-D CAD object and is analogous to a key in a relational database. Figure 5.3 shows the lists BOX-INDEX and BOX-ACTIVE that maintain the example version hierarchy and identify the state of each member version. Version b-1a2 is *suspended* as it is neither *declared* nor *active*.

We can extend the example version hierarchy by executing the following sequence of operators:

1. `activate`(BOX, b-1a2): Specifies version b-1a2 as *active*. Version b-2 which was initially *active* is now specified as *suspended*.

2. `declare`(BOX, b-1a2): Specifies version b-1a2 as *declared*.

3. `derive`(BOX, b-1a2): Creates a new *active* version b-1a3 and links it, as a child, to the previously *declared* version b-1a2. At the time of derivation, the contents of the parent version b-1a2 are logically copied into the child version b-1a3.

Table 5.1 summarizes the above sequence of operations. In the resulting BOX entity hierarchy, version b-1a3 is *active*, while version b-1a2 is now *suspended*. Versions b-1, b-1a0, b-1a1, and b-1a2 are in the *declared* state. Furthermore, version b-1a3 has the same description as its parent version b-1a2.

### 5.2.2 Version Change Management

We introduce a *check-out/check-back-deltas* protocol that structures the interaction between CAD applications and the version hierarchy. To modify the description of an *active* version, the designer *checks-out* out its `materialized` description into the CAD application.

Instances contained in that version description can be modified within the application environment using built-in drawing and editing tools. Most CAD systems provide several operators to modify 3-D CAD instances, we abstract these operators into one of `insert`, `delete`, and `replace` primitive operations. At the end of the given session, we detect the net changes made during that session as a set of *equivalent operations*, at most one on each instance that was modified during that given session. Two aspects of the framework makes this possible. First, we store, with each instance, the sequence of all operators that were executed on that instance. Second, a `compress` operator summarizes the operator sequence on each instance to determine the set of *equivalent operators* during that session. Finally, we *check-back* the detected changes into the version hierarchy; they are then `integrated` with the existing description of the *active* version. Furthermore, a version, in our model, is a unit of granularity containing the set of *equivalent operations* of all changes that were made while that version was *active*. We can also `compute` the net changes between an *ancestor-descendant* pair of versions in an entity hierarchy as the set of *equivalent operations* that can be executed on the concerned *ancestor* version to describe its *descendant*.

Our prototype implementation realizes the above change management processes by the following five basic operators:

- `materialize`: describes a version as the set of changes that logically belong to its definition.

- `translate`: maps a CAD drawing or editing command to its corresponding primitive operation. For each modified instance, we maintain a sequence of primitive operators executed during the particular CAD session.

- `compress`: determines the *equivalent primitive operation* on each modified instance by summarizing its associated sequence of primitive operations.

- `integrate`: merges the equivalent primitive operations on an entity with the description of its *active* version.

- `compute`: determines the difference between an *ancestor-descendant* pair of versions as a minimum set of changes or *deltas* that can be executed on the *ancestor* version to describe the *descendant*.

Procedures to implement these operators have been developed in Chapter 3 (Section 3.5). While the algorithms were developed for a relational data model, they are easily translated

Table 5.2: Example Sequence of Operators on a BOX Instance 2.78672e+06 (ENTITY-OPS-LOG List)

| *Handle-id* | Sequence of Operators |
|---|---|
| 2.78672e+06 | `replace` $\prec$ `replace` $\prec$ `replace` |

to the CAD scenario. The rest of this section demonstrates these operators using a single BOX instance (identified by *Box-id* 2.78672e+06) contained in *active* version b-1a3 of the "ongoing" example BOX entity derivation hierarchy. As mentioned in the introduction, this example has been tested on the prototype implementation of the model.

- `materialize`(BOX, b-1a3): We describe a version in terms of all instances that are logically belong to its definition; component instances can be either physically associated with that version, or logically *inherited* from an *ancestor* version. A `materialize` procedure retraces the derivation path of the current version till the most immediate *complete ancestor* version, and collects the latest *equivalent operation* on each instance. If the *equivalent operation* is not a `delete` operation then it is included in the version description. Figure 5.4 shows an instantiated description of newly `derived` version b-1a3 of the example BOX entity hierarchy. In this particular situation, the root version b-1 is the most recent *ancestor* for version b-1a3, that is specified as *complete.* Version b-1a3 does not contain any of the instances that belong to its definition; instance 2.78672e+06 is *inherited* from version b-1a2, while instances 2.44213e+06, 2.68377e+06, 2.78338e+06 and 2.67366e+06 are *inherited* from *ancestor* version b-1a0.

- `translate`: A `translate` function abstracts a CAD drawing or editing operation into one of the three primitive operations: `insert`, `delete`, or `replace`. For example, moving a BOX instance 2.78672e+06, of our "running" example, corresponds to a primitive `replace` operation that transforms the instance from its initial to final location. We maintain the list of primitive operators that were executed on each instance during a given session. Table 5.2 shows an ENTITY-OPS-LOG list containing an example sequence of three primitive `replace` operators on the BOX instance 2.78672e+06.

Table 5.3: Detected Change on BOX Instance 2.78672e+06 (Application Example)

| *Handle-id* | *Lx* | *Ly* | *Lz* | *X* | *Y* | *Z* | *Operator* |
|---|---|---|---|---|---|---|---|
| 2.78672e+06 | 321 | 6 | 144 | 308 | -250 | 0 | `replace` |

- `compress`: A `compress` operation determines the *equivalent operation* on each modified instance by summarizing its associated sequence of primitive operators. For each instance, a `compress` operation recursively substitutes the first two nodes of the operator sequence with its *equivalent operation*, till the entire operator sequence has been reduced to a single node, producing the *equivalent operator* on the concerned instance. Corollary 1 (Section 3.3.3) forms the basis for describing the detected *equivalent operations*. By this result, the inserted or replacement values of instances correspond to their descriptions at the end of a CAD session. On the other hand, replaced or deleted values were originally *checked-out* from the version hierarchy. The resulting *equivalent operations* are the net changes made during a particular CAD session, and are *checked-back* into the entity's *active* version. Table 5.3 gives the primitive `replace` operation on BOX instance 2.78672e+06 obtained by `compressing` the example ENTITY-OPS-LOG list (Table 5.2). The replacement value is the description of the BOX instance at the end of the given session.

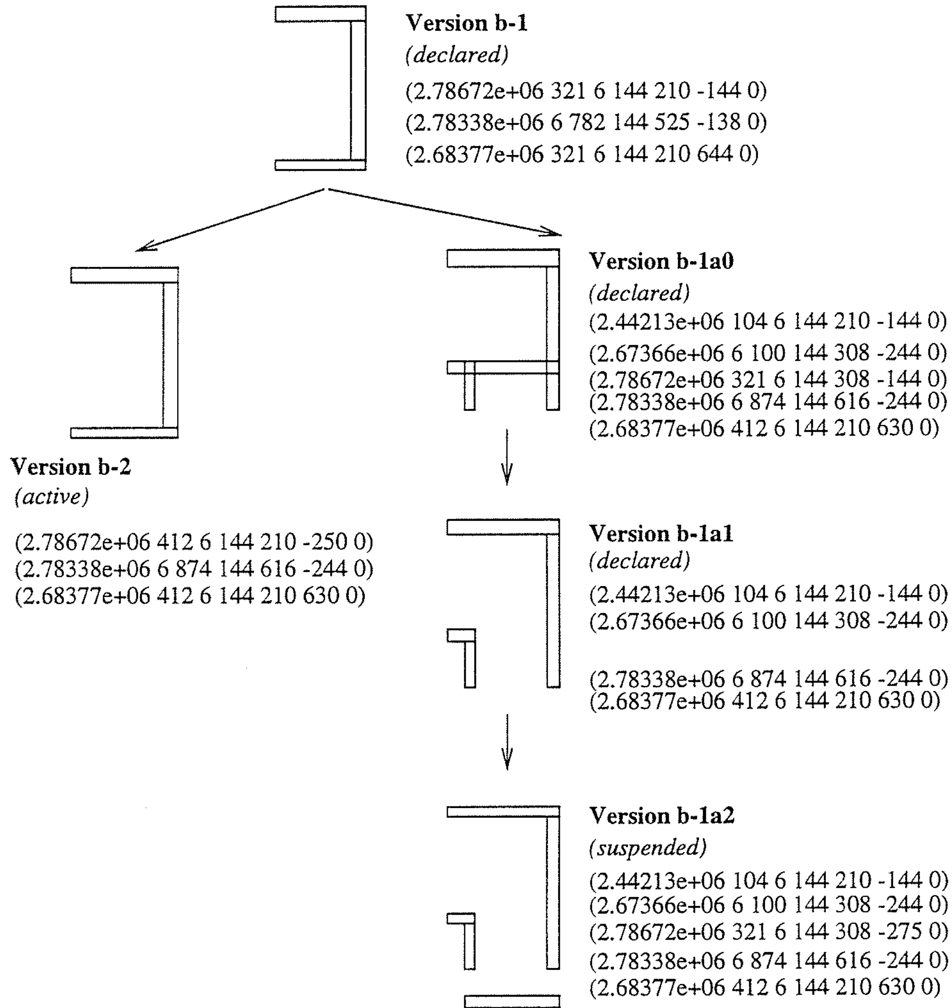
- `integrate`: An `integrate` operation merges the *checked-back* changes on an entity with the existing description of that entity's *active* version. As mentioned in Section 3.5.3, a given change on an instance has a *matching list* if that instance exists in the *active* version. Two possible situations exist for a change to have a *matching list*:

  - The instance description is physically associated with the *active* version.
  - The instance description is logically *inherited* from the *active* version's *ancestors*.

  We call the former a *physical matching list*, and the latter a *logical matching list*. Section 3.5.3 presented two criteria for `integrating` a change into an *active* version, and outlines an algorithm to implement the `integrate` operation. This procedure ensures that a version is represented by the set of *equivalent operations* of all changes that were *checked-back* by CAD sessions, while that version was *active*. The versioning scheme is therefore a *forward deltas* scheme [36]; a version can be described by

executing its associated set of *equivalent operations* on the description of its parent version. We illustrate an `integrate` operation by *checking back* the previously computed equivalent `replace` operation on BOX instance 2.78672e+06 (shown in Table 5.3) into the *active* version b-1a3 of our example version hierarchy. Figure 5.5 shows the resulting description of the BOX entity hierarchy.

- `compute`: A `compute` operation determines the changes between two versions, where one is an *ancestor* of the other in an entity derivation hierarchy. These computed changes are a minimal set of data operations or *deltas* that can be executed on the *ancestor* version to produce a description of the *descendant* version. The `compute` procedure is an innovative application of *equivalent operations*. The version model views a derivation path between an *ancestor-descendant* pair of versions as a set of *equivalent operator* sequences, one sequence for each instance that was modified in versions along the derivation path. By this interpretation, a `compute` operation involves determining the net *equivalent operation* for the operator sequence on each modified instance. We illustrate a `compute` operation by determining the differences between versions b-1 and b-1a3 of our example version hierarchy (Figure 5.5). The `computed` differences are expressed in terms of `insert`, `delete`, and `replace` operations that can be executed on version b-1 to produce a description of version b-1a3. Figure 5.6 shows the inserted BOX instances, while Figure 5.7 displays both the replaced as well as the replacement values of instances that have been modified along the concerned derivation path. For each instance, the `compute` operation determines the *delta* by considering only the first and last *equivalent operations* on that instance along the derivation path from version b-1 and b-1a3 (versions b-1a0, b-1a1, b-1a2, and b-1a3).




Scheme: < Box-id, Lx, Ly, Lz, Xcoord, Ycoord, Zcoord >

Figure 5.2: Example Version Hierarchy of a BOX Entity

```
(b-1 ((VERSION-TYPE COMPLETE) (PARENT-ID NULL) (DECL Y) (FR N) (PUB N)))
(b-2 ((VERSION-TYPE COMPLETE) (PARENT-ID b-1) (DECL N) (FR N) (PUB N)))
(b-1a0 ((VERSION-TYPE INCOMPLETE) (PARENT-ID b-1) (DECL Y) (FR N) (PUB N))
(b-1a1 ((VERSION-TYPE INCOMPLETE) (PARENT-ID b-1a0) (DECL Y) (FR N) (PUB N
(b-1a2 ((VERSION-TYPE INCOMPLETE) (PARENT-ID b-1a1) (DECL Y) (FR N) (PUB N
(b-1a3 ((VERSION-TYPE INCOMPLETE) (PARENT-ID b-1a2) (DECL N) (FR N) (PUB N

(b-1a3)
```

Figure 5.3: Final Representation of the Version Hierarchy (BOX-INDEX and BOX-ACTIVE Lists)

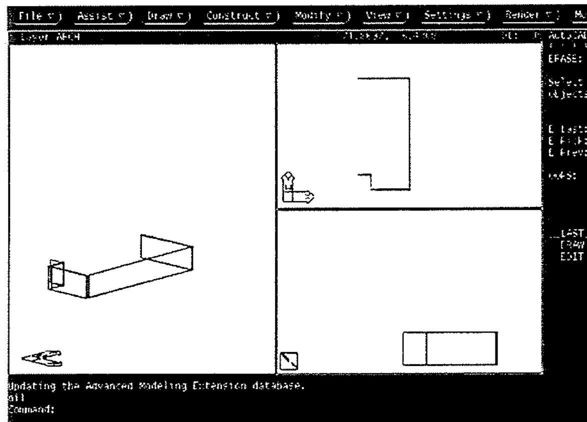



Figure 5.4: Instantiation of Version b-1a3 in Box Derivation Hierarchy (`materialize`(b-1a3))

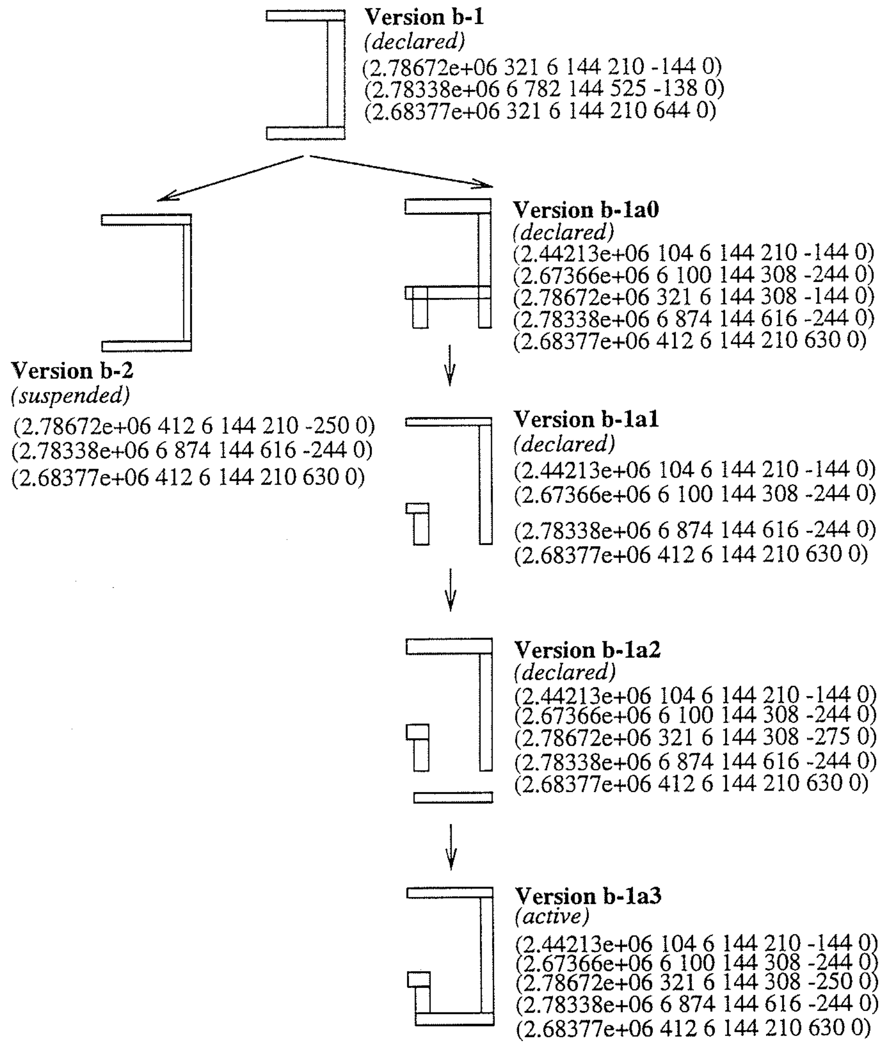



Figure 5.5: Description of Box Entity Hierarchy after Modifying *Active* Version b-1a3

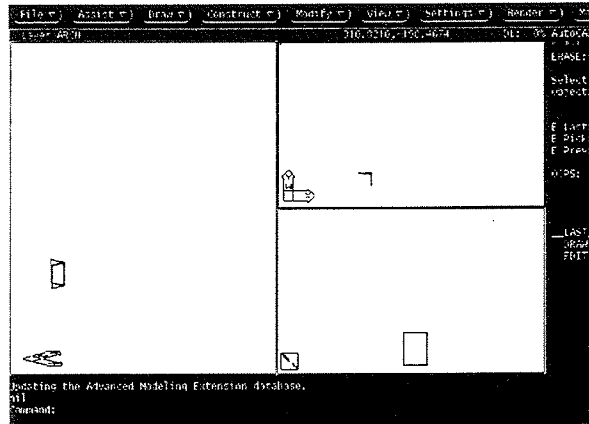



Figure 5.6: Computation of Changes between Versions b-1 and b-1a3 (`inserted` Operations)

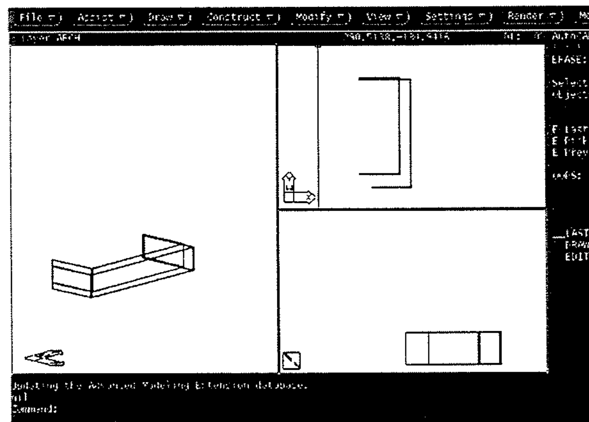



Figure 5.7: Computation of Changes between Versions b-1 and b-1a3 (`replace` Operations)

## 5.3 Assembly Model

There are two important differences between the implementations of the assembly model in the relational and CAD environments. Firstly, in the CAD environment, an individual component of an assembly can be an instance of either another complex entity or a primitive entity. While the former is represented by an assembly, the latter is an instance contained in a specific version in the entity derivation hierarchy. We recursively expand an assembly definition to develop a *component hierarchy.* The root of the hierarchy is the original assembly, while leaf nodes correspond to primitive instances *included* in the assembly definition. Specifically, an instance of a primitive entity is said to be *included* in an assembly if it is either a component of that assembly definition or is *included* in one of its components. We denote versions that contain primitive instances *included* in an assembly definition as *included* versions.

The second distinction between relational and CAD implementations of assemblies is that, in addition to the **union** composite modeling operation, the CAD prototype provides **intersect** and **subtract** operators. These operators are in turn supported by AUTOCAD's Advanced Modeling Extension (AME). These two assembly characteristics are motivated primarily by the way CAD systems are used in current design situations to build complex entity descriptions. Extensions to the assembly scheme are possible because in a CAD environment, an entity scheme is less formally defined; unlike the relational data model, we need not specify a new entity scheme (create a new relation) each time we perform a new composite modeling operation.

The rest of this section is organized as follows: The first part outlines a scheme to represent assemblies in a CAD environment. An assembly is defined as a composite modeling operation on a set of components. The second part of this section discusses the operators used to specify the status and access properties of an assembly definition. The proposed algorithms extend previously presented procedures as they first generate a component hierarchy for the given assembly, and ensure that the property values being assigned to the assembly definition are shared by each of its *included* instances. Finally, the third part of this section discusses operations to manage changes among assemblies. We describe an assembly as a composite modeling operation on its components, and characterize changes between an *ancestor-descendant* pair of assemblies, in terms of the changes between their individual components.

### 5.3.1 Representation Scheme

A scheme to implement the assembly model must represent both the *composition* and *evolution* relationships among assemblies. While a *composition* relationship identifies instances *included* in an assembly's *component hierarchy*, an *evolution* relationship identifies an earlier description of the complex entity from which the current assembly has been `generated`.

We maintain assemblies in a particular discipline D as a list D-ASSEMBLY. An assembly in this list is uniquely identified by a system-generated attribute, *Assembly-id*. Each assembly contains a set of its component instance identifiers, *Components*. If a particular component is a primitive entity, it is referenced as a specific instance belonging to a particular version in the entity derivation hierarchy. On the other hand, a component complex entity is located as an assembly in that concerned discipline. In addition, we store the identifier of any parent assembly, *Parent-id*, from which the current description of the complex entity could have been `generated`.

We now present the definition of the list, D-ASSEMBLY:

( (<u>*Assembly-id*</u> Value)<br>
  ( (*Operator* Value)<br>
    (*Components* Value $)_{\{j \mid j = 1, c\}}$<br>
    (*Parent-id* Value$)))_{\{l \mid l = 1, w\}}$

which has $w$ assemblies in the discipline D. The remaining attributes are specified as follows:

- *Assembly-id:* System generated attribute that uniquely identifies an assembly in a particular discipline.
- *Operator:* Descriptor of the *composite modeling operation* that is executed on the component instances of an assembly to produce its final description. Possible values are "union," "intersect," and "subtract."
- *Components:* Set of $c$ instances that are components of the assembly definition. A component primitive instance is uniquely identified as an instance belonging to a version in a particular entity derivation hierarchy. We can thus identify such a component instance by the attributes: *Entity-Name*, *Version-id* and a set of *primary attributes*. Similarly, a component complex instance is identified as an assembly with identifier, *Assembly-id*, in the given discipline.

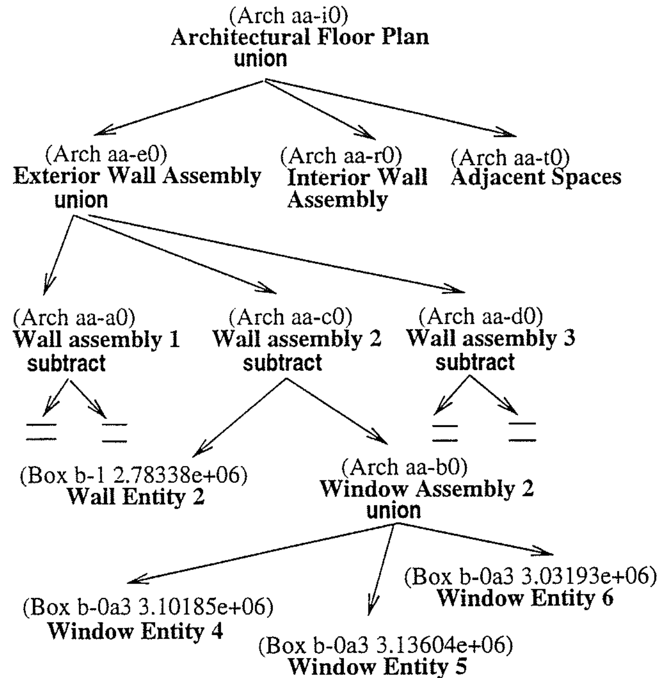



Figure 5.8: Component Hierarchy of Initial Architectural Layout (Assembly aa-i0)

- *Parent-id:* Identifier of the parent assembly from which the current assembly has been `generated`. In situations where the assembly was `defined` independently this attribute is assigned a value, "Null."

The initial architectural layout of the example Medical Cyclotron facility (shown earlier in Figure 5.1) is represented by a *total* assembly, aa-i0. Figure 5.8 shows a partial component hierarchy for this floor plan, that is formed by the **union** of three assemblies that represent (i) exterior walls, (ii) interior walls, and (iii) an adjoining facility. Assembly aa-e0 which denotes the set of exterior walls is in turn formed by the **union** of individual wall assemblies. Further, a particular exterior wall assembly aa-c0 is formed by **subtracting** window and door openings (assembly aa-b0) from the solid wall. This wall corresponds to the Box instance 2.78338e+06 in version b-1 of the entity hierarchy given in Figure 5.5. Remaining elements in the example *component hierarchy* of assembly aa-i0, including other exterior walls, interior walls and the adjacent facility, can be similarly expanded.

### 5.3.2 Modified Implementation of Assembly State Operators

We modify the procedures presented earlier to assign status and access properties for an assembly definition. Specifically, we consider the `freeze-assembly`, `archive-assembly`, and `publish-assembly` operators. The modifications to these procedures have been necessitated by the following two factors: (i) an assembly, in a CAD environment, can be further composed of other assemblies as well as instances of primitive entities, (ii) the close coupling of the version and assembly models requires assembly status or access property values to be shared by each of its components. To satisfy these requirements, we adopt the following approach to assign particular assembly properties.

1. Expand the initial assembly definition to develop its *component hierarchy.*

2. Check that all *included* versions, and intermediate assemblies in the generated component hierarchy share the desired assembly property. For intermediate assemblies or *included* versions which do not currently share the required property values, the assembly operator invokes pertinent state operators. Importantly, an assembly operator fails if even one of its *included* versions cannot be assigned the required property values.

3. Assign the desired property value to the assembly definition provided that all elements in the component hierarchy satisfy the minimum property requirements.

Figures 5.9, 5.10 and 5.11 outline procedures to `freeze`, `publish`, and `archive` assemblies. These procedures generate the component hierarchy in a post-order fashion, ensuring that each *included* version has the minimum property values needed. In the event of failure, the current implementation does not **rollback** the entire operation. It does, however, maintain a list of the successfully executed version and assembly state operators, giving designers the information needed to manually restore the database to its original condition, if so desired. However, as will be shown by an example later, the designers often re-execute the original assembly operators after correcting the causes of failure.

In addition to the data types provided in previous chapters, the proposed procedures to `freeze`, `archive`, and `publish` assemblies use the following additional data type:

- component: Generic type declaration of a component of an assembly. A component could be either an instance of a primitive entity or another assembly.

We also employ the following external functions in developing the given assembly state operators. These functions are in addition to those provided in the previous chapters.

- `first-component` = (discipline D, $id_{assembly}$ A) component C: Returns the first component C of an assembly A in the given discipline D. The function creates a new pointer, Comp-locator, which maintains the last referenced component of the given assembly.

- `next-component` = (discipline D, $id_{assembly}$ A) component C: Accesses the component C of assembly A in discipline D that is subsequent to the one that was most recently referenced, as determined by the current position of a Comp-locator pointer. Successful execution of a `next-component` function advances the pointer by one component.

- `type` = (component C) string S: Determines if a given component C is an instance of a primitive entity or an assembly describing a complex entity. The type of a component is expressed as a character string S. Possible values are "primitive," and "complex."

- `inc-entity` = (component C) entity E: Returns the entity name E of the component C which is of type, "primitive."

- `inc-version` = (component C) $id_{version}$ V: Returns the identifier of the version V that is an element component C which is of type, "primitive."

- `inc-assembly` = (component C) $id_{assembly}$ A: Returns the identifier of the assembly A that is an element of component C which is of type, "complex."

- `inc-entity` = (component C) entity E: Returns the entity name E of the component C that is of type, "primitive."

Note the procedures to `thaw` and `suppress` assemblies, which were presented in Section 4.1.5, are unaffected by the changes to the assembly scheme. This is because `thaw` and `suppress` operations do not alter status and access properties of their components, as a component version could be *included* in other *frozen* or *published* assemblies as well. We use a design situation from our "ongoing" Cyclotron facility design example to illustrate the `publish-assembly` and `freeze-assembly` operators. To better meet the client's requirements, the architect independently rearranged the initial layout of the facility that was previously presented in Figure 5.1. Figure 5.12 shows this modified floor plan. This floor plan

is represented by assembly aa-i1 which is `generated` from the initial layout, assembly aa-i0. Figure 5.13 gives the *component hierarchy* for assembly aa-i1. In assembly aa-i0, the facility's exterior walls are represented by instances belonging to version b-1 of the BOX entity. In assembly aa-i1, however, these BOX instances now belong to version b-1a3. From Figure 5.5 (Section 5.2.2), note that version b-1a3 is a *descendant* of version b-1 and is currently in the *active* state. Realizing possible effects of this layout change on other designers, the architect invokes a `publish-assembly` operator to provide others access to the new assembly aa-i1. However, the assembly model requires that an assembly definition must be *frozen* or *archived* before it can be `published`. The architect therefore executes a `freeze-assembly` operation on assembly aa-i1. As *included* BOX version b-1a3 is *active*, it cannot be assigned a status value, "freeze," causing the entire `freeze-assembly` operation on assembly aa-i1 to fail. The architect overcomes this failed attempt by `declaring` version b-1a3 and then re-executing the `freeze-assembly` operator. A subsequent `publish-assembly` operation makes the new facility design, aa-i1, accessible to the entire design team.

### 5.3.3 Assembly Change Management

We manage changes among assemblies along both the *composition* and *evolution* links. Along a *composition* relationship, we instantiate the description of an assembly as the result of a composite modeling operation on its components. We implement a `display-assembly` operator that generates an assembly *component hierarchy*, and recursively aggregates the composite modeling operations on `materialized` descriptions of its *included* instances to describe the original assembly. Figure 5.14 shows an example shear wall-framing system (assembly sa-f0) designed by the structural engineer for the initial architectural floor plan (Figure 5.1).

Along an *evolution* relationship, we characterize changes between an *ancestor-descendant* pair of assemblies by recursively aggregating the `computed` changes between corresponding pairs of instances that are *included* in both assemblies. The current prototype has implemented this procedure using a `characterize-assembly-deltas` operator. Using this operator, a structural engineer, in our example, can easily determine the changes between a newly `published` architectural layout, aa-i1, and a previously `published` layout, aa-i0, that she is currently referencing. The effect of determining the changes between these two *published* layouts is equivalent is similar to an *asynchronous communication* of the net design changes by the architect to the structural engineer. In this model, however, the structural

engineer has access to the entire set of computed changes, and is left with the responsibility of identifying the smaller subset of changes that impact her framing system.

**Algorithm 37** (`freeze-assembly`)

*Input: d:* Name of the concerned discipline.

*v:* Identifier of a *defined* assembly that is being assigned a status property value, "freeze."

*Output: success:* Boolean indicator of a successful operation.

```
procedure freeze-assembly = (discipline d, id_assembly a) boolean success:
begin component c, entity e, id_version v;
  success := FALSE;
  c := first-component(a);
  while (c ≠ nil) do
    if (type(c) = "primitive") begin
      e := inc-entity(c);
      v := inc-version(c);
      success := freeze(e, v);
    else
      a := inc-assembly(c);
      success := freeze-assembly(d, a);
    end;
  end;
  if (success) begin
    state-assembly(d, a) := frozen;
  end;
end procedure
◇
```

Figure 5.9: Procedure to `freeze` a *Defined* Assembly in a Particular Discipline

**Algorithm 38** (`publish-assembly`)

*Input: d:* Name of the concerned discipline.

*a:* Identifiers of the *frozen* or *archived* assembly being `published`.

*Output: success:* Boolean indicator of a successful operation.

```
procedure publish-assembly = (discipline d, id_assembly a) boolean success:
begin component c, entity e, id_version v;
  success := FALSE;
  c := first-component(a);
  while (c ≠ nil) do
    if (type(c) = "primitive") begin
      e := inc-entity(c);
      v := inc-version(c);
      success := publish(e, v);
    else
      a := inc-assembly(c);
      success := publish-assembly(d, a);
    end;
  end;
  if (success) begin
    if (state-assembly(d, a) = frozen) begin
      state-assembly(d, a) := published;
    end;
    if (state-assembly(d, a) = archived) begin
      state-assembly(d, a) := persistent;
    end;
  end;
end procedure
◇
```

Figure 5.10: Procedure to `publish` a *Frozen* or *Archived* Assembly

**Algorithm 39** (archive-assembly)

*Input:* *d:* Name of the concerned discipline.

*a:* Identifier of a *frozen* or *published* assembly being archived.

*Output:* *success:* Boolean indicator of a successful operation.

```
procedure archive-assembly = (discipline d, id_assembly a) boolean success:
begin component c, entity e, id_version v;
   success := FALSE;
   c := first-component(a);
   while (c ≠ nil) do
      if (type(c) = "primitive") begin
         e := inc-entity(c);
         v := inc-version(c);
         success := archive(e, v);
      else
         a := inc-assembly(c);
         success := archive-assembly(d, a);
      end;
   end;
   if (success) begin
      if (state-assembly(d, a) = frozen) begin
         state-assembly(d, a) := archived;
      end;
      if (state-assembly(d, a) = published) begin
         state-assembly(d, a) := persistent;
      end;
   end;
end procedure
◇
```

Figure 5.11: Procedure to archive a *Frozen* or *Published* Assembly

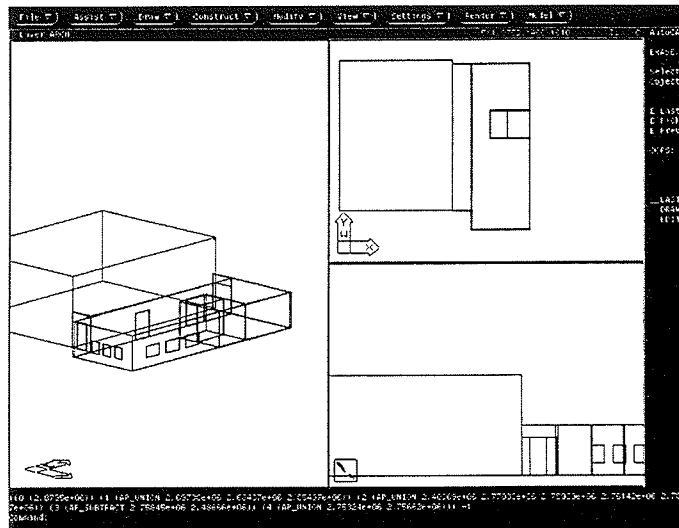


Figure 5.12: New Floor Plan of the Facility (Assembly aa-i1)

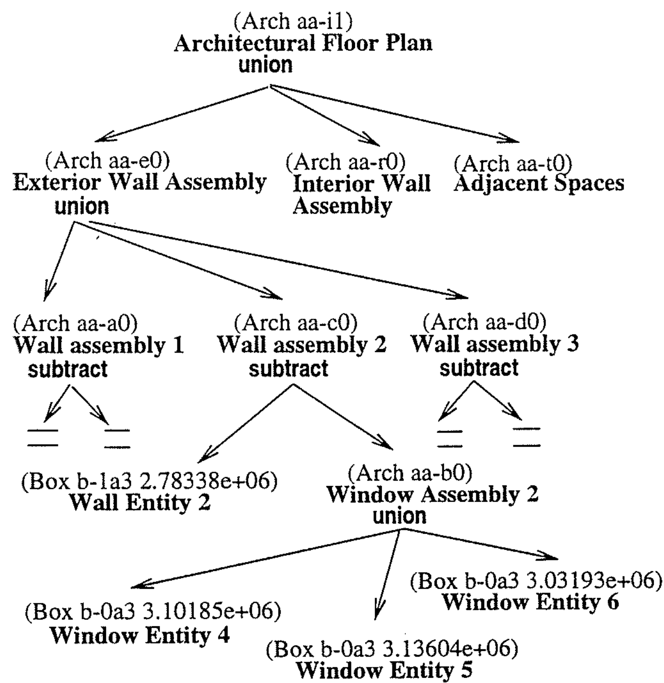



Figure 5.13: Component Hierarchy of New Architectural Layout (Assembly aa-i1)

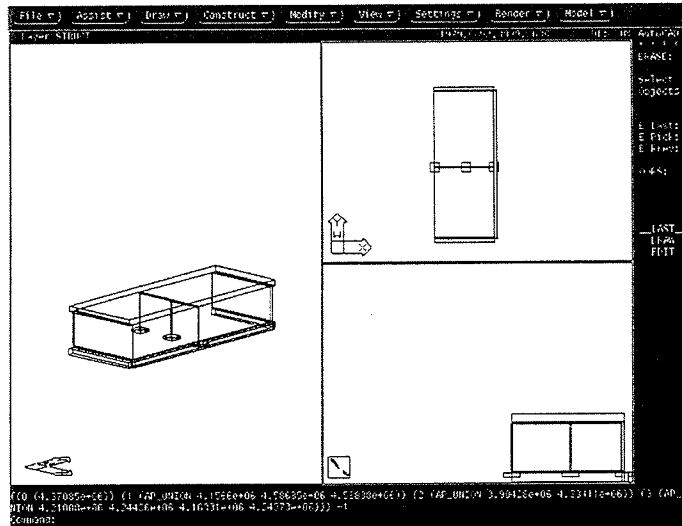


Figure 5.14: Current Structural System (Assembly sa-f0) Based on Initial Floor Plan

## 5.4 Configuration Model

In this section, we first discuss a scheme to implement the configuration model in a CAD paradigm. Two types of relationships are considered, *composition* and *evolution.* A *composition* relationship establishes the individual designs from the participating disciplines that compose the project description; an *evolution* relationship establishes the link between a configuration and its parent from which the configuration has been `generated`. In addition, we specify access and status properties for configurations which simulate collaborative environments. Secondly, we describe operators that manage changes among project designs. The close coupling of versions, assemblies, and configurations allows us to describe changes between project designs in terms of changes between their component assemblies (and in turn their *included* versions) from each of the participating disciplines.

### 5.4.1 Representation Scheme

We propose a scheme to represent both the *composition* and *evolution* relationships among configurations. This scheme also describes the property values for configurations belonging to that discipline. We maintain configurations created in a discipline, D, as a list, D-CONFIGURATION. A particular configuration, in this discipline, is represented by a list that is uniquely identified by a system-generated attribute, *Config-id*. *Composition* relationships are established by associating with each configuration a set of its component *total* assemblies, *Components*. On the other hand, *evolution* relationships identify an earlier project description from which the current configuration has been `generated`. Such links are stored in a *Parent-id* attribute. The configuration scheme also maintain the assigned status and access property values in attributes *Fr-status* and *Pub-status*, respectively. We now present a definition of the D-CONFIGURATION list:

( (*Config-id* Value)
  ( (*Components* Value $)_{\{i \mid i=1,d\}}$
  (*Parent-id* Value)
  (*Fr-status* Value)
  (*Pub-status* Value)$))_{\{l \mid l=1,g\}}$

which maintains $g$ configurations in a discipline D. The other specified attributes are:

Table 5.4: Example Configurations Created by a Structural Engineer

| Config | Arch. Assembly | Struct. Assembly | Hvac Assembly | *Parent* |
|---|---|---|---|---|
| sc-1 | aa-i0 | sa-f0 | ha-a0 | "Null" |
| sc-2 | aa-i1 | sa-f0 | ha-a0 | sc-1 |

- *Config-id:* System generated identifier that uniquely identifies a configuration created by a designer from discipline D.
- *Components:* Set of component *total* assemblies, one from each of the *d* participating disciplines.
- *Parent-id:* Identifier of any parent configuration from which the current configuration has been `generated`. In situations where the configuration was `defined` independently, this attribute is assigned a "Null" value.
- *Fr-status:* Indicator of the status property value of a configuration. Possible values are "y"("freeze"), "a" ("archive"), and "n" ("not freeze").
- *Pub-status:* Indicator of the access property value of a configuration. Possible values are "y"("publish"), and "n" ("not publish").

Table 5.4 shows two configurations created by a structural engineer of our example facility design scenario. In this example, configuration sc-1 is `defined` in terms of an architectural layout (assembly aa-i0), a structural framing system (assembly sa-f0) and a mechanical ducting system (assembly ha-a0). Also, configuration sc-2 is `generated` as a descendant of configuration, sc-1. The architectural layout aa-i0 in configuration sc-1 is replaced in configuration sc-2 by its *descendant* assembly, aa-i1.

### 5.4.2 Support for Project Change Management

We manage changes among configurations along both the *composition* and *evolution* links. Along a *composition* relationship, we instantiate a configuration description in terms of its components. The current prototype has implemented this functionality as `display-configuration` operator that invokes `display-assembly` operators to describe designs from

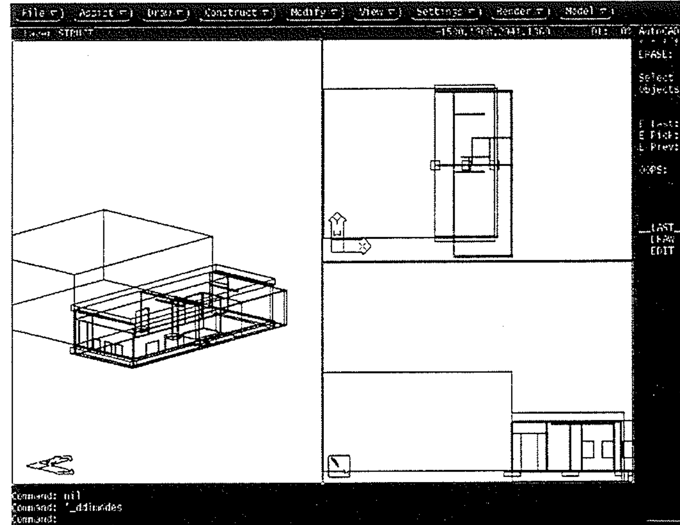


Figure 5.15: *Intermediate* Configuration sc-2 to Validate Current Structural System Against New Floor Plan

the participating disciplines. Figure 5.15 shows the example *intermediate* configuration sc-2 of our Cyclotron facility design example as inconsistent. On the other hand, along an *evolution* relationship, we determine the changes between two configurations, where one is an *ancestor* of the other. These changes are represented in terms of changes between corresponding pairs of component designs from the same discipline. We implement a `characterize-config-deltas` operator that executes `characterize-assembly-deltas` operations to compute the changes between each pair of component assemblies from the participating disciplines. As mentioned in Section 4.1.6, a `characterize-assembly-deltas` operator in turn determines the differences between two design descriptions in terms of the `computed` changes between each corresponding pair of instances *included* in the two assemblies. The `characterize-config-deltas` operation enables a project leader to monitor the overall progress of a project, both efficiently and accurately. Since the computation of the project changes is in terms of changes between individual designs, the project leader can further determine the relative progress of the design process in the different participating disciplines.

## 5.5 Summary and Conclusions

In this chapter, we have presented the three-layered data management model of versions, assemblies, and configurations in a CAD environment. We have implemented the scheme in a AUTOCAD environment, and have validated the change management capabilities of the model using a real though simplified design example. The purpose is to demonstrate that our model is comprehensive in that it can handle the independent evolution of various kinds of data found in design situations.

We believe that design data is mostly of three types: (i) alpha-numeric (such as engineering properties of design entities), (ii) graphical (such as the geometry and spatial arrangements of the design entities), and (iii) textual (such as design specifications and progress reports). This situation is further complicated as the various kinds of data reside in different environments. While alpha-numeric data is typically stored in traditional databases (network, relational or object-oriented), graphical objects are primarily stored in CAD environments. Textual data is usually unstructured and can be found in ASCII files, which may be linked to word processing applications.

A number of research efforts have attempted to model spatial relationships in a relational environment [29]. However, there is a mismatch between the more formal structure of relational schemes and the flexibility required for manipulating CAD drawings, making these modeling efforts cumbersome and computational intensive. The proliferation of CAD systems in recent years motivates the management of evolving CAD drawings.

As mentioned in Chapter 1, a number of schemes have been developed to version text files in software engineering environments. Similarly, schemes have been proposed for evolving alpha-numeric data in the form of design attribute values. However, we have not encountered any scheme that manages evolving design drawings within the CAD environments in which they are situated.

Through this chapter, we have established that the proposed data management model supports 3-D CAD objects, just as it handles relational tuples. With this validation, we assert that the model provides a comprehensive solution for project data management. Furthermore, we have implemented the model in a CAD environment, proving that our theoretical three-layered framework of versions, assemblies and configurations is independent of the underlying data model by which the design entities are expressed.

The schemes, developed in this chapter, to represent the three-layered data management

model in the CAD environment closely parallel the schemes proposed earlier (Chapters 3 and 4) for the relational environment. An important exception is the scheme to represent assemblies. Most CAD systems provide assemblies as a framework to compose simpler geometric objects into more complex forms. These systems are flexible in allowing composite entities to be described without previously specifying relationships among their component instances. However, commercial CAD systems neither provide efficient version control mechanisms for primitive entities nor do they capture temporal information for composite entities. Versions, in our prototype, maintain evolving descriptions of the primitive graphical entities such as the BOX entity. We use the standard composite modeling operations, union, intersect, and subtract to form complex entities in terms of their components. Importantly, however, our assembly truly adds a time dimension (or more precisely, a design history) to CAD drawings by representing assemblies in terms of component instances that belong to different versions. Finally, configurations aggregate *total* assemblies from each discipline to describe an overall project. Though the current implementation is centralized, we simulate a distributed environment using layers provided by AUTOCAD, and a perspec operator to switch focus among them. Note that the model itself is independent of the architecture of its computer implementation, and can be easily implemented in a distributed paradigm.

Finally, we show the change management capabilities of our model as applied to a simple design example. We apply the concept of *equivalent operations* to store, detect, and manage changes among versions in an entity derivation hierarchy. The close coupling of the three layers of the model allows us to characterize changes along the *composition* and *evolution* relationships at the assembly, as well as configuration levels. In this chapter, we outline procedures to implement this capabilities in a CAD system. Through a scripted example on a Medical Cyclotron facility example, we illustrate how these concepts can be applied for coordinating and monitoring activities in typical design projects.

# Chapter 6

# Summary and Conclusions

In this thesis, we have developed a data management model for collaborative design environments. More specifically, we address the storing and managing of changes among designers in a multidisciplinary project. The model has three salient features. First, we propose a three-layered model of *versions*, *assemblies*, and *configurations*. *Versions* maintain evolving descriptions of primitive entities within a single discipline. *Assemblies* integrate component instances to describe more complex entities, as well as designs within individual designs. *Configurations* provide a framework to represent an overall project design which is composed of designs from the participating disciplines. Second, we introduce the concept of an *equivalent operation* for a valid sequence of changes that is a single data operation which results in the same final description of the instance as the original sequence of changes. We prove that a valid sequence of changes can be summarized into at most one *equivalent operation* which is also valid. Further, we can uniquely determine this *equivalent operation* from just the first and last elements in the original sequence. Third, we apply *equivalent operations* to the three-layered framework. *Equivalent operations* form the theoretical basis for storing, detecting, and managing changes on versions of an entity. The close coupling of the version, assembly, and configuration layers enables the computed version changes to be combined for characterizing changes at the assembly and configuration levels. Using these three concepts, the model efficiently supports **project coordination** through the *asynchronous communication* of changes among designers, as well as **project monitoring** through systematic tracking of evolving project descriptions.

## 6.1 Model as a Comprehensive Data Management Solution

We assert that the proposed model is a comprehensive solution for project data management. We base this belief on our understanding of collaborative design environments. Two fundamental premises are crucial to our discussion.

Firstly, in multidisciplinary projects, designers work independently on various aspects of the project, while sharing information as necessary for coordination. The four basic version states, *active*, *suspended*, *declared*, and *removed*, are sufficient to support the design process in an individual discipline. Branching in a particular version hierarchy allows designers to maintain several solution alternatives. By `activating` *suspended* versions, the designer can switch focus among the various alternatives, independently developing them in parallel.

Assemblies provide a mechanism to aggregate instances in component versions to describe more complex entities (*partial* assemblies), as well as complete designs in an individual discipline (*total* assemblies). We provide access and status properties for assemblies to promote cooperation on a project. By `publishing` a *total* assembly, a designer shares the design description with other consultants, both within and outside of the design team. Importantly, the designer retains control over the information being shared. This is critical in professional design environments where despite their cooperative spirit, designers typically insist on controlling the information they provide others. Their need for autonomy arises from a number of factors. For one, designers from different disciplines are often affiliated with different organizations (or different departments within the same organization). They also maintain a varied agenda on the project according to their professional training and responsibilities. In our framework, the close coupling of the assembly and version models requires that the properties of an assembly definition are shared by each of its component versions.

Configurations provide a framework for designers to manage the shared information. Formally, a configuration describes a multidisciplinary project design as an aggregation of design descriptions from each of the participating disciplines. We provide basic status and access properties for configurations which simulate environments that facilitate cooperation. Based on its particular properties, we classify a configuration definition into one of four states. Using an *intermediate* configuration, a designer can privately evaluate his/her design with respect to the entire project. *Accessible* configurations simulate meeting scenarios in which each designer brings his/her design to the table, allowing the entire team

to collectively evaluate the design progress. *Landmark* configurations represent project descriptions that are maintained for extended periods and are accessible to members both within and outside the design team. These include (i) team records checkpointing project descriptions at the end of specific design phases, (ii) project designs submitted to regulatory agencies for construction approval, and (iii) documents released to contractors for bidding purposes. Finally, *recorded* configurations allow designers to maintain, for personal references, alternative designs that were not selected for the current project.

Our second premise about multidisciplinary design projects is that engineering design data is basically of three types: (i) alpha-numeric (such as engineering properties of design entities), (ii) graphical (such as the geometry and spatial arrangements of design entities), and (iii) textual (such as design specifications and progress reports). Moreover, we realize that the different types of data reside in disparate environments. For example, alphanumeric design data is typically stored in traditional databases (network, relational, or object-oriented), while graphical objects are primarily stored in CAD environments. Textual data is usually unstructured and can be found in ASCII files which may be linked to word processing applications.

By implementing the data management model in a relational, as well as a CAD environment, we establish that the model is independent of the underlying data model used for representing the design description. This reasserts that the proposed model is a comprehensive solution for multidisciplinary design projects. Importantly, the implementation of the model avails the specific features of that environment.

A case in point is the implementation of the assembly model in the CAD environment. CAD systems traditionally use assemblies for recursively composing complex geometric forms from simpler components. The implementation of our model in the CAD environments uses this flexibility to extend the relational implementation of assemblies in two ways. Firstly, in the CAD environment, an individual component of an assembly can be an instance of either another complex entity or a primitive entity. While the former is represented by an assembly, the latter is an instance contained in a specific version in an entity derivation hierarchy. We recursively expand an assembly definition to develop a *component hierarchy*. The root of such a hierarchy is the original assembly, while leaf nodes correspond to primitive instances *included* in the assembly definition. Specifically, an instance of a primitive entity is said to be *included* in an assembly if it is either a component of that assembly definition, or is *included* in one of its components. We denote versions that

contain primitive instances *included* in an assembly definition as *included* versions.

The second distinction between relational and CAD implementations of assemblies is that, in addition to the **union** composite modeling operator, the CAD prototype provides **intersect** and **subtract** operators which are supported by AUTOCAD's Advanced Modeling Extension (AME). These two extensions were feasible because in a CAD environment, an entity scheme is less formally defined than the relational data model; we need not specify a new entity scheme (create a new relation) for each distinct complex entity formed by a new composite modeling operation.

## 6.2 Limitations and Future Work

As mentioned in Section 1.2, we have classified design data management problems into three basic categories: (i) information transfer, (ii) understanding design intent, and (iii) storage requirements. This research has primarily addressed issues in storing evolving project design data. Extensions to the model must also maintain the underlying rationale that guide the design process. This research partially addresses the transfer of design information through the asynchronous communication of detected design changes. For realistic scenarios, we must integrate this asynchronous scheme with more robust synchronous communication methodologies that support greater interaction in tightly-coupled design situations. Future efforts must incorporate mechanisms that address the following issues: (i) capturing and disseminating design intent, (ii) detecting inconsistencies that potentially arise due to specific design changes, and (iii) identifying and notifying the impacted design team members.

In the more limited context of generalizing the present framework, future efforts should be directed towards improving the computational efficiency of the proposed algorithms, expanding the scope of the implementation, and validating the scalability of the model using more involved design examples. We describe each of these research directions.

1. *Generalizing change control algorithms:* Some of the procedures presented in this thesis apply to fairly specific situations. Such procedures can be generalized to make them more applicable in a wider variety of design situations. A specific example is the `characterize-assembly-deltas` procedure. While determining the net changes between an *ancestor-descendant* pair of assemblies, the current procedure assumes that the *component hierarchies* of the two assemblies have the same structure. Future implementations can incorporate more general algorithms that relax this requirement.

2. *Improving the computational efficiency of the change control operations:* The current implementations are computationally inefficient as they do not avail mechanisms, such as indexing strategies, to improve their performance. As mentioned earlier, the current implementations primarily serve to validate the proposed concepts. The change control algorithms given in this thesis must be revisited, incorporating schemes that increase their efficiency. While relational systems provide indexing schemes, current CAD systems deal mostly with ASCII pile files. We hope that this work coupled with the proliferation of CAD systems in design environments motivates CAD vendors to address this limitation. Also, the change detection procedures which are currently built on top of a CAD system must be incorporated within CAD environments. This is essential for realistic implementations of our concepts in large scale projects.

3. *Implementing the model in distributed environments:* One of our basic premises is that designers typically work independently, while sharing information necessary for collaboration. This premise is truly realized in distributed environments. Our current CAD implementation though centralized simulates distributed situations using layers that are provided in the AUTOCAD environment. A future project would involve truly distributing the implementation across several hardware and software options and supporting this networked environment.

4. *Validating the scalability of the model:* An obvious next step is to test this model for more complex design environments till it is stable in real-world projects. Such an effort is unrealistic in an academic research project, but is essential for the acceptability of the proposed model.

In summary, we believe this thesis is exploratory in nature, outlining a comprehensive data management model that manages project changes both across disciplines and through various levels of detail. Further, we have developed implementations in both relational and CAD environments to demonstrate its potential application in real project scenarios.

# Bibliography


[1] *Report from The 1984 Workshop on Advanced Technology for Building Design and Engineering.* National Academy Press, Washington D.C., 1984.

[2] *Report from The 1985 Workshop on Advanced Technology for Building Design and Engineering.* National Academy Press, Washington D.C., 1985.

[3] *Report from The 1986 Workshop on Advanced Technology for Building Design and Engineering.* National Academy Press, Washington D.C., 1986.

[4] R. Agrawal and H. V. Jagdish. On Correctly Configuring Versioned Objects. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 367–374, Amsterdam, Netherlands, 1989.

[5] V. Ambriola, L. Bendix, and P. Ciancarini. The Evolution of Configuration Management and Version Control. *Software Engineering Journal*, pages 303–310, 1990.

[6] N. Belkhatir and J. Estublier. Experiences with a database of programs. *ACM SIGPLAN Notices*, 22(1):84–91, 1987.

[7] W. Cellary and G. Jomier. Consistency of Versions in Object Oriented Databases. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 432–441, Brisbane, Australia, 1990.

[8] H. Chou and W. Kim. A Unifying Framework for Version Control in a CAD Environment. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 336–346, Kyoto, Japan, 1986.

[9] S. I. Feldman. Make - A Program for Maintaining Computer Programs. *Software Practice and Experience*, 9:255–265, 1979.

[10] S. J. Fenves et al. An Integrated Software Engineering Environment for Building Design and Construction. In *Proceedings of the Fifth ASCE Computing in Civil Engineering Conference*, pages 21–32, Alexandria, VA, 1988.

[11] B. J. Gray et al. Establishing Requirements for Data Management Control Applications to Achieve a Successful System Implementation and Configuration. *Proceedings, 1989 ASME International Computers in Engineering Conference and Exposition*, pages 1–9, 1989.

[12] A. N. Habermann and D. Notkin. Gandalf software development environments. *IEEE Transactions on Software Engineering*, 12:1117–1127, 1986.

[13] K. Hall. A Framework for Change Management in a Design Database. Thesis STAN-CS-91-1379, Department of Computer Science, Stanford University, 1991.

[14] H. C. Howard et al. Versions, Configurations, and Constraints in CEDB. Working Paper 031, Center for Integrated Facility Engineering, Stanford University, 1994.

[15] H. C. Howard and D. R. Rehak. KADBASE: Interfacing Expert Systems with Databases. *IEEE Expert*, 4(3):65–76, 1989.

[16] J. W. Hunt and M. D McIlroy. An Algorithm for Differential File Comparison. Computing Science Technical Report 41, Bell Laboratories, 1976.

[17] R. H. Johnson. Engineering Data Management - What's Needed and Expected for the 1990's. *Proceedings, 1989 ASME International Computers in Engineering Conference and Exposition*, pages 17–22, 1989.

[18] R. H. Katz. Toward a Unifying Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):374–408, 1990.

[19] R. H. Katz et al. Version Modeling Concepts for Computer-Aided Design Databases. In *Proceedings of the ACM SIGMOD Conference*, pages 379–386, Washington D.C., 1986.

[20] R. H. Katz et al. Design Version Management. *IEEE Design and Test*, 4(1):12–22, 1987.

[21] A. M. Keller and J. D. Ullman. A Version Numbering Scheme with a Useful Lexicographical Order. In *Proceedings of the IEEE Data Engineering Conference*, pages 240–248, Taipei, Taiwan, 1995.

[22] A.M. Keller. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. In *Proceedings of the Fourth ACM Sigact-Sigmod PODS Symposium*, pages 154–163, Portland, Oregon, 1985.

[23] M. V. Ketabchi and V. Berzins. Modeling and Managing CAD Databases. *IEEE Computer*, 20(2):93–102, 1987.

[24] T. Khedro, M. R. Genesereth, and P. M. Teicholz. Agent-Based Framework for Integrated Facility Engineering. *Engineering with Computers*, 9:94–107, 1993.

[25] P. Klahold, G. Schlageter, and W. Wilkes. A General Model for Version Management in Databases. In *Proceedings, Twelfth International Conference on Very Large Data Bases*, pages 319–327, Kyoto, Japan, 1986.

[26] K. Krishnamurthy and R. Fruchter. Feedback on Observation of Communication among Design Consultants in Systemix Project. Working Paper 21, Center for Integrated Facility Eengineering, Stanford University, 1992.

[27] K. Krishnamurthy and K. H. Law. Configuration Management in a CAD Paradigm. In *Proceedings of the International Mechanical Engineering Congress and Exposition*, pages 103–116, San Francisco, CA, 1995. ASME.

[28] G. S. Landis. Design Evolution and History in an Object Oriented CAD/CAM Database. In *31st COMPCON Conference*, pages 297–305, San Francisco, CA, 1986.

[29] K. H. Law and M. K. Jouaneh. Data Modeling for Building Design. In *Proceedings of the Fourth ASCE Computing in Civil Engineering Conference*, pages 21–36, Boston, MA, 1986.

[30] D. B. Leblang and R. P. Chase. Parallel software configuration management in a network environment. *IEEE Software*, pages 28–35, 1987.

[31] F. Londono et al. A Blackboard Scheme for Cooperative Problem Solving by Human Experts. In D. Sriram et al., editors, *Computer Aided Cooperative Development.* Springer-Verlag, 1991.

[32] R. A. Lorie and W. Plouffe. Complex Objects and their Use in Design Transactions. In *Proceedings of the ACM Design for Engineering Applications Database Week*, pages 115–121, San Jose, CA, 1983.

[33] R. Morenc and R. Rangan. Information Management to Support Concurrent Engineering Environments. In *Proceedings of the 1992 ASME International Computers in Engineering Conference and Exposition*, pages 135–148, San Francisco, CA, 1992.

[34] H. Peltonen et al. An Engineering Document Management System. In *Proceedings of the ASME Winter Annual Meeting*, New Orleans, LA, 1993.

[35] B. Prasad et al. Information management for concurrent engineering: Research issues. *Concurrent Engineering: Research and Applications*, 1:3–20, 1993.

[36] M. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[37] D. Spooner and M. Hardwick. Using Persistent Object Technology to Support Concurrent Engineering Systems. In P. Gu and A. Kusiak, editors, *Concurrent Engineering: Methodology and Applications*, pages 205–234. Elsevier Science Publishers B.V., 1993.

[38] D. Sriram. Computer Aided Collaborative Product Development. Research Report R91-14, Intelligent Engineering Systems Laboratory, Massachusetts Institute of Technology, 1991.

[39] P. T. Swinehart et al. A Structural View of the Cedar Programming Language. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, 1986.

[40] W. F. Tichy. RCS - A System for Version Control. *Software Practice and Experience*, 15(7):637–645, 1985.

[41] J. D. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, Rockville, Maryland, 1989. Vols. 1 and 2.