Version and Configuration Management for Collaborative Design

by
Karthik Krishnamurthy

TECHNICAL REPORT
Number 92

November, 1993

Stanford University
Copyright © 1993 by
Center for Integrated Facility Engineering

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 94305-4020
Summary

Title: Version and Configuration Management for Collaborative Design

Author: Karthik Krishnamurthy, Department of Civil Engineering.

Date: November 1993

Funding: Research Fund from the National Science Foundation, Grant No. IRI-9116646. Funded by CIFE as a CIFE seed research project.

Abstract

This report proposes a model for the management of data for collaborative facility design. This model proposes a versioning scheme to manage design data in a given discipline. In addition, this model proposes configurations as a mechanism to describe an overall project design as an integration of the designs from each individual discipline. Additional properties are specified for a configuration definition to support cooperation among designers.

Subject

This report describes a model that describes the management of design data in a given discipline, and also provides a framework to integrate the designs to describe the overall project. This model includes a versioning scheme to manage design data in a given discipline. A version specifies the state of a design corresponding to a particular discipline. The versioning scheme is hierarchical in structure and supports the evolution of multiple design alternatives independent of each other. In addition, this model proposes configurations as a mechanism to describe an overall project design as an integration of the designs from each individual discipline. Designs from different disciplines represent different facets of the same physical entity, and therefore interact with each other. The restrictions due to these interactions are specified as constraints. In other words, a configuration is defined as a set of component versions (one from each participating discipline) and an associated set of constraints. Additional properties are defined for a configuration definition to support cooperation among designers. The properties of a configuration place restrictions on its component versions. This report enumerates these inclusion dependencies and discusses the model of versioning to support the configuration model.

Objective / Benefits

Design is a multi-disciplinary activity. A model for the management of project design data must address the data management requirements in each discipline. In addition, the model must provide a framework to integrate the designs in individual disciplines to describe the overall project. The objectives of this research is to develop a model that can:

- manage multiple design alternatives for each discipline, and support the incremental addition of data to each alternative.
- maintain a history of previous designs, and represent intermediate designs from which a designer could redesign.
• provide a framework to integrate the designs in each discipline to describe the overall project design. The interactions among the designs in the different disciplines must also be specified.

• support the evaluation of the project design to detect inconsistencies due to interactions among the designs, which may be notified and later resolved through negotiation.

• maintain project designs for use by design team members, as well as, by consultants outside the design team. These designs may also be stored for reference in later projects.

The main benefit of this research is the development of a comprehensive model to support the data management needs for collaborative design, that addresses both the requirements of design applications in an individual discipline, as well as, the integration of these designs to describe the design of the overall project.

Methodology

To comprehend the data management needs for collaborative design, a design project for remodeling a facility was observed. These observations illustrated the limitations of the approaches currently adopted by designers to manage design data and motivated the requirements of a model for data management for collaborative design. Previous research efforts in version and configuration management, as well as, systems for collaborative design were surveyed. The model is developed in three incremental steps. In the first step, a model of versions is proposed to support design transactions. A simple transaction model using a check out / check in protocol is considered. The second step proposes a model of versions to support configurations. Configurations are formalized as a framework to integrate design states in different disciplines. Restrictions are placed on versions included in a configuration definition. The versioning model is extended to support these inclusion dependencies. The last step proposes a model of versions to support collaboration. Additional properties are defined for configuration states to support cooperation among designers. The additional restrictions on the component versions is supported by this version model.

Results

This report evaluates the proposed model as a solution to the data management needs for collaborative design. In addition, the results of this research are briefly enumerated:

• Configurations are proposed as a conceptual framework to integrate designs in different disciplines to describe an overall project design. Additional properties are associated with a configuration definition to support cooperation among designers.

• A model of versions is proposed to manage design data in a particular discipline. This scheme supports the independent development of multiple design alternatives. In addition, the model supports the restrictions placed on versions due to the properties of a configuration definition in which it may be included.

• In this model, version states are distinguished on the ability of a design transaction to modify its contents.

• The incremental addition of data to a design version is supported by implementing versions in terms of deltas. This report, however, does not describe the implementation model developed for the versioning scheme.
• The model considers an *instance-instance* inheritance scheme along the *descendant-of* relationship. Also, this scheme uses a static binding mechanism.

A very simple application example is presented to illustrate the proposed model. A prototype implementation describing this example is expected to be completed by December 1993.

**Research Status**

This report describes the current state of the model under development. Extensions proposed to the current model include:

• Extension of the model to support dynamic inheritance schemes.

• Formalization of the detection of changes made by the execution of a transaction interacting with the versioning scheme through a check out / check in protocol.

• Development of a recursive configuration model to support multiple levels of assemblies of entities.

The expected date of demonstration of the implemented prototype for the model proposed is June 1994.
Abstract

Design is a multidisciplinary activity. A model for the management of project design data must address the data management requirements in each discipline. In addition, the model must provide a framework to integrate the designs in individual disciplines to describe the overall project. In this report, a model is proposed for the management of configurations and versions as a framework for data integration among the design disciplines. This model includes a versioning scheme to manage design data in a given discipline. A version specifies the state of a design corresponding to a particular discipline. The versioning scheme is hierarchical in structure and supports the evolution of multiple design alternatives independent of each other. In addition, this model proposes the use of configurations as a mechanism to describe an overall project design as an integration of the designs from each individual discipline. Designs from different disciplines represent different facets of the same physical entity, and therefore interact with each other. The restrictions due to these interactions are specified as constraints. In other words, a configuration is defined as a set of component versions (one from each participating discipline) and an associated set of constraints. A designer may define a configuration by using a version from his/her own discipline and a version from each of the other disciplines.

Additional properties are defined for a configuration definition to support cooperation among designers. Configuration states are distinguished on the properties of the configuration definition. The state of the configuration places restrictions on its component versions. Thus, the execution of an operator on a configuration state may result in changes in the restrictions specified on its component versions. In addition, preconditions may be specified on the execution of operators on versions included in a configuration definition. This report enumerates these inclusion dependencies and discusses the model of versioning to support the configuration model.
Acknowledgements

The contents of this report have been influenced by the input of my entire doctoral committee. I am most grateful to Prof. Kincho Law, my principal advisor, who has spent long hours helping me crystallize ideas into the current form of this report. Formulation of ideas into a concise framework has been a long and arduous task and would have been much more difficult without his guidance. He has also helped me maintain an engineering perspective while developing more abstract concepts. I would like to thank Dr. Arthur Keller for helping me develop an insight into the database issues of version and configuration management. His significant feedback has positively influenced the development of the models proposed. I would like to thank Prof. Paul Teicholz for constantly urging me to ground this work on a realistic scenario. I thank him for setting up the exposure to a design project to motivate this work. His constant encouragement and input have been extremely critical. Prof. Jeff Ullman has given me the benefit of his tremendous insights into the problem. He has focussed me on the most critical issues of the problem and has guided my approach to think about them. In short, I would like to thank my entire research committee and look forward to their continued guidance and support. I would also like to thank Ashish Gupta and Sanjai Tiwari for their assistance throughout this project. It has been a strong team effort. I am very grateful to the design team members on the Systemix project who gave us the benefit of their design experience. Mr. Randall Dowler of Ehrlich-Rominger gave us a lot of his time and attention despite a very busy schedule. I am also grateful to Ms VanKeuren, Mr. Patrick Man and Mr. Jim Lentfer of Rinne & Peterson for their time and interest.

This research has been supported by the National Science Foundation, Grant No. IRI-9116646.
Contents

Abstract

Acknowledgements

1 Introduction
1.1 Observations of a Design Scenario ............................................. 1
1.2 Requirements of a Model for Collaborative Design .......................... 2
1.3 Objective of Research ............................................................... 3
1.4 Architecture of Data Management System .................................... 3
1.5 Organization of the Report ......................................................... 5

2 Related Work
2.1 Survey of Relevant Systems ....................................................... 6
2.2 Survey of Related Version and Configuration Schemes .................... 6
2.2.1 Scenario .................................................................................. 6
2.2.2 Organization of Version Set of an Individual Entity .................... 7
2.2.3 Properties of Version States .................................................... 7
2.2.4 Version Grouping .................................................................... 8
2.2.5 Relation Among Design Entities ............................................. 8
2.2.6 Binding Mechanisms ............................................................... 8
2.2.7 Representation of a Configuration .......................................... 8
2.2.8 Inheritance ............................................................................ 9
2.2.9 Change Management ............................................................. 9
2.3 Discussion .................................................................................. 9
2.3.1 Research Issues Surveyed ..................................................... 9
2.3.2 Research Issues Addressed ................................................... 10

3 Versioning Scheme to Support Application Transactions ................ 11
3.1 Application Transaction Model .................................................. 11
3.2 Version Model to Support Transactions ....................................... 13
3.2.1 Operators on the Data Contents of Versions ............................. 14
3.3 Dependencies Between Check In and Version State Operations .......... 14

4 Versions to Support Configurations ............................................. 16
4.1 Configuration Model ................................................................... 16
4.1.1 Operators on the Contents of a Configuration ........................... 18
4.2 Version Model to Support Configurations .................................... 18
5 Versioning to Support Collaborative Design
   5.1 Configuration Model to Support Collaborative Engineering .................. 22
   5.2 Version Model to Support Collaboration ........................................... 23
   5.3 Configuration Inclusion Dependencies .............................................. 23

6 Application Example
   6.1 Illustration of the Versioning Scheme ............................................. 28
   6.2 Illustration of the Configuration Model ........................................... 31

7 Summary and Conclusions
   7.1 Evaluation of the Proposed Model .................................................. 37
      7.1.1 Support for Data Management Needs of a Design Application in a Single
            Discipline ................................................................................. 37
      7.1.2 Support for Collaboration and Integration of Designs in Multiple Disciplines 38
   7.2 Current Status of Research .............................................................. 38
   7.3 Future Work ..................................................................................... 38

Bibliography ......................................................... 40
List of Tables

4.1 Specification of Version States due to Inclusion Dependencies .......................... 21
5.1 Specification of Version States due to Inclusion Dependencies (Extended Configuration States) .......................................................... 27
6.1 Operations on Version States (Application Example) .......................................... 29
6.2 Operations on Version Contents (Application Example) ...................................... 32
6.3 Specification of Version States (Application Example) ...................................... 32
List of Figures

1.1 Assumptions for a Distributed Data Management System 4
1.2 Architecture of a Data Management System for Collaborative Design 4

3.1 Application Transaction Model 12
3.2 Model of Versioning to Support Design Transactions 13
3.3 Dependency of Version State Operators on Application Transactions 15

4.1 Model of Configurations 17
4.2 Model of Versions to support Configurations 19

5.1 Model of Configurations to support Collaborative Design 24
5.2 Model of Versions to support Collaborative Design 25

6.1 Application Example Scenario 29
6.2 Initial Version Hierarchy (Beam Entity) 30
6.3 Final Version Hierarchy (Beam Entity) 33
6.4 Computing Deltas between Versions (Beam Entity) 34
6.5 Final Version Hierarchy (Crane Entity) 35
6.6 Configuration Definitions (Application Example) 36
Chapter 1

Introduction

In this report, a model is proposed for version and configuration management for collaborative facility design. Engineering design is a multi-disciplinary activity. The design process for each discipline is carried out independently by consultants in that field. The designs generated in the individual disciplines are integrated to describe the overall design of the project. The designs in the different disciplines interact with each other. These interactions may result in inconsistencies among the individual designs. These inconsistencies must be detected early in the design process. They may then be resolved by the concerned designers through negotiation. This report proposes a model that supports the management of design data in a given discipline, and also provides a framework to integrate the designs to describe the overall project.

The proposed model is applied to the design of engineering facilities. The design disciplines considered include architectural, structural, and mechanical engineering. To comprehend the data management needs for collaborative design, a design project for remodeling a facility was observed. The observations made during this exposure are presented in Section 1. The requirements that the model needs to satisfy are summarized in Section 2. The objectives of this research effort are stated in Section 3. The overall architecture of a data management system to support collaborative design is described in Section 4. The last section presents an outline of the rest of this report.

1.1 Observations of a Design Scenario

A collaborative design project was observed to understand the limitations in the approaches currently adopted by designers for the management of design data. In the project observed, the design team comprised of designers from different consulting organizations. Our observations of this collaborative design effort are summarized as follows:

1. Designers in each discipline independently developed their design drawings. These drawings were then compiled to describe the overall project design.

2. Each designer developed multiple design alternatives, and continuously added more details to each alternative, i.e., each design alternative evolved as the design progressed.

3. Unsatisfactory modifications to a design prompted designers to revert to earlier designs from which they either redeveloped the design or generated other alternatives.
CHAPTER 1. INTRODUCTION

4. Designs in different disciplines interacted with each other. Individual designers have the knowledge of interactions among the designs. Usually, they were aware of the effects of their design modifications on designs in other disciplines. However, this process of detecting potential inconsistencies was ad-hoc.

5. A designer selected an alternative based, not only on considerations from his domain, but also on the ease with which it integrated with designs in other disciplines.

6. Design team members were notified of decisions in other disciplines that could potentially affect their design. A notification was initiated either by a design modification or by the need for data from other disciplines. Thus, the process of detecting potentially conflicting modifications and communicating them was ad hoc and resulted in delays.

7. Meetings provided a forum for the designers to evaluate the interactions among the designs in different disciplines. Inconsistencies that were detected needed to be resolved before the designs could be integrated. Meetings were more critical in the early stages of the design process when the interactions among the designs in different disciplines were not well understood and the ramifications of specific decisions were unclear. However, as the design progressed, the interactions were better understood, and designers were able to individually notify the specific designers who were affected. Also, any detected inconsistencies were resolved among the concerned designers. In the schematic phase, the designers used the meetings as a platform to evaluate their designs from the perspective of the overall project. Additionally, the entire design team participated in the evaluation of the progress of the entire project.

8. The outcome of each important design phase was a set of project drawings describing the overall state of the project at the end of that phase. Some of these designs were made available to consultants outside the design team, such as the city council, as well as, contractors for purposes of approval, contract bidding and cost estimation.

The observations described above motivated the specification of the requirements of a model for data management for collaborative design. These requirements are enumerated in the next section.

1.2 Requirements of a Model for Collaborative Design

The model for the management of data for collaborative design must address the requirements for the design in a particular discipline, as well as, provide a framework to describe the overall project design. The model must address the following requirements to support the data needs for a collaborative design environment:

- The model must manage multiple design alternatives and should support the incremental addition of data to each alternative.

- The model must maintain a history of previous designs. These may represent intermediate designs from which a designer may wish to redesign.

- The model must provide a framework to integrate the designs in each discipline to describe the overall project design. The interactions among the designs in the different disciplines must also be specified.
• The model must support the evaluation of the project design to detect inconsistencies due to interactions among the designs. The affected designers may be notified of the detected inconsistencies. These inconsistencies may then be resolved through negotiations.

• The model must maintain project designs for use by the design team members, as well as by consultants outside the design team. These designs may also be stored for reference in later projects.

This research effort proposes a hierarchial versioning scheme to support the evolutionary nature of designs in a discipline. A version represents a specific state of a design alternative in a particular discipline. The versioning scheme manages multiple design alternatives and also supports the incremental addition of data to each version. Configurations are proposed as a framework to integrate versions from different disciplines to describe a project design. The following section explicates the objectives of this research effort.

1.3 Objective of Research

The objective of this research is to develop a model for data management to support collaborative design. The model needs to support design applications in each discipline. In addition, it must provide a mechanism to integrate the designs in different disciplines. A versioning scheme is proposed to support design application transactions in a given discipline. Restrictions on designs due to the interactions among the disciplines are represented as constraints. A configuration is, thus, specified as a set of versions (one from each discipline) and an associated set of constraints. Although the evaluation of the configuration is outside the scope of this effort, the model supports mechanisms for that purpose. Also, notification schemes must be supported that communicate any constraint violations to the affected designers. However, the actual notification mechanism and associated protocols are outside the scope of this research.

The model is to be implemented in a distributed database environment, where each site supports design applications in a given discipline. The versioning scheme at each site must support the incremental addition of data to a given version (specific design state). Constraints associated with a configuration are evaluated by a constraint checker. The particular implementation used utilizes an incremental constraint checking methodology[15]. To effectively support the above requirements, the versioning scheme is implemented by managing design changes (deltas). This model requires the detection of changes made to a design by the execution of a design transaction. A check out / check in protocol is proposed for the interaction of an application transaction with the database at a given site. In this work, the Discipline Product Models are specified using the Structural Data Model [9]. This along with the mathematical grounding of the relational data model motivated the selection of relational databases as data repositories at each node. To focus on the inter-disciplinary parameters, a single application is considered for each discipline at a given site. The assumptions are summarized in Figure 1.1.

1.4 Architecture of Data Management System

The overall architecture of the data management system to support collaborative design is presented in Figure 1.2. The Version Manager adopts a hierarchical versioning scheme to support design applications at the given site. The Configuration Manager integrates versions at different database sites to describe the project design. The project design is evaluated by a Constraint
CHAPTER 1. INTRODUCTION

Figure 1.1: Assumptions for a Distributed Data Management System

Figure 1.2: Architecture of a Data Management System for Collaborative Design
Manager. The configuration is incrementally evaluated against a specified base configuration[36]. The configuration manager specifies the changes between the current configuration and the specified base configuration. These changes are characterized in terms of the changes in the corresponding component versions and the associated set of constraints. Violations that may result from the evaluation are returned to the Configuration Manager. The violations are, then, referenced to the design data it affects. The Notification Manager notifies the violations to relevant designers. The design changes made by an executed application transaction are computed by a Change Detection Module. These changes are checked into the versioning scheme on the successful commitment of the transaction. The interaction of a design application with the database at that site is specified by a check out/ check in protocol.

1.5 Organization of the Report

The rest of this report presents the logical foundations of the proposed version and configuration management system. Previous research efforts are surveyed in the next chapter. These include systems developed for collaborative design, as well as, research efforts in version and configuration management. A model of versions to support design transactions in a given discipline is proposed in Chapter 3. A simple transaction model using a check out/ check in protocol is presented. A model of versions to support the data needs of the transaction model is also proposed in this chapter. Chapter 4 formalizes configurations as a framework to integrate design states in different disciplines. Dependencies that result from including a version in a configuration definition place additional requirements on the versioning scheme at a site. A model of versions to support configurations is developed. Additional properties are defined for a configuration definition to support cooperation among designers in different disciplines. The properties of a configuration place additional restrictions on its component versions. A model of versions to support collaborative design is also proposed in Chapter 5. A simple application example is presented in Chapter 6 to illustrate the model of versions and configurations developed in this report. The last chapter evaluates the contributions of this work as a solution to the problem of data management for collaborative design. The issues in implementing the model and future research directions are also summarized in this chapter.
Chapter 2

Related Work

This chapter summarizes some previous systems developed for collaborative engineering as well as research efforts in versioning and configuration management. The last section discusses the impact of these efforts in relation to the objectives of this work.

2.1 Survey of Relevant Systems

Different systems have been developed for data and information management for collaborative design [35][29][11][5]. These systems address two basic issues: (i) Project Design Data Management, and (ii) Information Transfer among designers.

- Project Design Data Management: Most of the systems surveyed have considered a centralized repository to store the final project design [35][5][11]. The DICE project [29] considers a distributed environment with each agent having its own local database, although a centralized data store is used to store the overall design. KADBACE [17] primarily addresses the issue of semantic and syntactic translations for data retrieval in a heterogeneous distributed environment.

- Information Transfer: The systems surveyed have proposed centralized mechanisms to post design notifications (Blackboard schemes [11][35], Facilitator[25]).

2.2 Survey of Related Version and Configuration Schemes

Most of the work in version management addresses the following issues[18]: (i) Organization of version set of an individual entity, (ii) Properties of version states, (iii) Relationship among the engineering entities, (iv) Version of an entity in terms of the versions of the entities from which it is assembled, (v) Nature of inheritance, (vi) Constraint and Change propagation.

Each research effort considers a specific scenario for which a version management system is presented. This section reviews related research based on the above issues.

2.2.1 Scenario

Most research efforts surveyed consider a hierarchy of repositories with at least the following three levels:
CHAPTER 2. RELATED WORK

- **Public Level**: The base level (Archive [20], Public [6], Parts [24]) stores data that is shared by the entire team of designers. Data in this repository is consistent and may not be modified or deleted by a designer. Additionally, only data that does not violate the consistency of the repository can be added to it.

- **Private Level**: Each designer maintains a repository whose contents may be accessed as well as modified. This repository has been referred to as Private [20][6][24]. Data in the designers' repository need not be consistent at any given point in time.

- **Group Level**: Most research efforts define an intermediate level, where the work in progress of two or more designers can be combined to evaluate if the assembly works as required. Data contained in this repository may be deleted by the owner but may not be modified. This level has been referred to as Group [20], or Project [6][24] workspaces. They differ in their requirements of consistency of contained data. While Katz [20] permits inconsistent data to be stored in the Group workspace, Chou and Kim [6] require consistency requirements in the Project workspace.

A different scenario is proposed by Ecklund et al.[8]. They consider a distributed scenario as a federation of databases. Designers could be either associate members or participating members at a given site. Participating members have read and write access to the data at the given site while Associate members may only be granted read access privileges.

In most of the surveyed works, transfer of data across workspaces is by the check out / check in paradigm [20][6][8][28], or some variation of it: acquire, modify, reconcile [34].

### 2.2.2 Organization of Version Set of an Individual Entity

A design entity evolves with the progress of the design process. Implicitly associated with the versioning scheme is the information about the derivation of versions. However, initial research efforts did not maintain this information [7][31][2]. Dittrich [7] considered a design object as a set of versions. Alternatives, revisions, and different representations were represented by an AND / OR graph by McLeod et al.[31]. A type-version hierarchy was proposed by Batory et al [2]. Beech and Mahbod [3] proposed version instances to be organized into version sets associated with a single generic instance. A version graph could then be realized via functions associated with generic and version instances. Katz et al. [21] represented version derivation history as a tree structure with is-derived-from links. Additionally, they specified a structural object generalized from the versions to specify a type definition. Some research efforts [26][8] generalized the representation of version derivation information by using a rooted DAG. Refinement, derivation, and consolidation were defined as the three relationships linking versions in [8]. Rochkind [32] proposed the SCCS system in which he defined a linear history of versions in terms of interleaved deltas organized as releases and levels. This system is enhanced by the SUN NSE system [34] and the Apollo DSEE system [28], which supports alternatives. Vines et al [37] proposed timestamps instead of version numbers to store temporal history of versions.

### 2.2.3 Properties of Version States

A notion of a current version is proposed in [21][7]. The property of currency is that only derivatives of the current version or its successors can be added to the version history. Ecklund et al [8] define
a principal path through the DAG. The current version of the object is then defined as the most recently created version on this path. [32] defines two special delta types: optional deltas and deltas to explicitly includes or excludes certain specified deltas. Optional deltas represent fixes for specific users. The exclusion facility enables the omission of mistakes, while inclusion enables a delta added later to be visible in releases later in the derivation history.

2.2.4 Version Grouping

Different mechanisms have been proposed to group related versions. Dittrich et al [7] proposed Version Clusters to group versions in an arbitrary way, while Klahold et al [26] advanced Partitions to provide a single level division of the version space, and proposed Subset Graphs to represent portions of the version graph that satisfy certain consistency criteria.

2.2.5 Relation Among Design Entities

Two kinds of relationships are defined among entities[20]: (i) Component Hierarchy (is-part-of relationship), and (ii) Equivalences (is-equivalent-to relationship). The former relationship is used to specify a composite entity in terms of its component entities. The latter relates different representations of the same physical artifact. In [20], the component hierarchy is defined as a DAG, where an entity can be a component of more than one composite entity. Equivalences link different entities that represent different aspects of the same engineering artifact. Haskin et al [16] augmented the relational model to support aggregates of tuples across relations. Batory et al [2] separated implementation from interface and defined a notion of molecular objects. The notion of equivalences has been referred to as template refinements[24].

2.2.6 Binding Mechanisms

A version may be bound to a specified version at compile time (static binding [23][2]), or may be dereferenced at run time to some version specified in the entity’s version history (dynamic binding [23][2]). Alternate terminology, such as fixed and floating references [8], or specific and generic references[3] have been used in literature. Different mechanisms are specified to dereference dynamic binding to specific versions. Katz [23] adopted layers [12] to store portions of version histories of different entities to be used together, and contexts to order them. Dittrich [7] used a notion of environments to accomplish the same task. An environment has direct entries which explicitly specify versions, indirect entries which reference other environments, and inclusion entries which prioritize referenced environments. Batory and Kim [2] specified parametrized versions to specify dynamic binding. Another approach was to resolve the binding to the current version [27][8]. Beech et al [3] specified a general mechanism based on database triggers which could be fired either before or after an operation invocation. In this case, the user specifies the rules for resolution.

2.2.7 Representation of a Configuration

A configuration is defined as the version of a composite entity in terms of the versions of the component entities [20][30][27][37]. Ketabchi et al[24] referred to the same notion as explosion refinement. The Apollo DSE system uses configuration threads to tabulate the component names[27].
2.2.8 Inheritance

Batory et al [2] proposed type-version inheritance as the basis of their version model. Katz et al [19] defined an instance to instance inheritance scheme to handle inheritance along the following relationships: descendant-of, component-of and is-equivalent-to.

2.2.9 Change Management

Techniques for Change Propagation as well as Change Notification have been proposed. Mechanisms such as Group Check In [23], Pended Versions [27], Change and Version sensitive relationships [37], Timestamp methods [2], and Propagation Attributes [33] have been proposed to disambiguate and limit the scope of the propagation of a change. Chou et al [6] proposed message based and flag based change notification schemes.

2.3 Discussion

The first subsection discusses the issues in version and configuration management that have been presented in the literature surveyed. The second subsection presents the objectives of this work emphasizing the points of departure from previous efforts.

2.3.1 Research Issues Surveyed

Research projects in versioning and configuration management have focussed on the following issues: (i) versioning of an individual entity, and (ii) versioning of an assembly of related entities in terms of the versions of the comprising entities, (iii) inheritance among versions, and (iv) implementation schemes. The main ideas proposed for each of the above issues is enumerated:

- **Versioning of an individual entity**: Version derivation histories have been formalized as a hierarchy [21] or, more generally, as a rooted DAG [26] [8]. In both situations the versions are connected by derived-from links.

- **Versioning of an assembly of entities**: Two kinds of relationships have been considered among entities[20]: component-of, and equivalent-to. Configurations have been defined as the version of the composite entity in terms of the versions of its components [20][24][27][30] [37].

- **Inheritance among versions**: Batory [2] proposed type-version inheritance, while Katz et al [19] proposed instance-instance inheritance schemes along descendant-of, equivalent-to, and component-of relationships. However Katz has explored only inheritance along the descendant-of relationship in detail.

- **Implementation Schemes**: Implementation of versions in terms of deltas supports the incremental addition of data to a version. However, this scheme has only been presented for the software engineering environment [32][28]. None of the versioning systems support the evolutionary nature of the design process, i.e., the incremental addition of data to an individual version.
2.3.2 Research Issues Addressed

The model proposed in this work supports the independent development of designs in different disciplines and their integration to describe the overall project design. None of the systems surveyed have addressed this specific problem. In developing this model, existing work has been adapted wherever possible. In addition, new proposals have been made that contribute to the field. The issues addressed in this work are presented explicating departures from existing work:

- A hierarchical versioning scheme for a single entity similar to the one proposed in [21] is considered at each site.

- Relationships among entities are described using the Structural Data model[9]. The component-of relationship may be specified by the ownership connection of the structural model. The subset connection in the structural model specifies a type-definition relationship. The equivalence relationship may be partially specified by the identity connection of the structural model. In addition, the structural model specifies a reference connection that relates design entities that interact with each other in different ways. The semantics of this connection has not been explored in previous works in versioning of an assembly of entities. Previous research efforts only considered the versions of composite entities in terms of the versions of its components. The current research aims to extend the binding mechanisms and change management techniques proposed to include versions connected by reference and subset connections. This would be necessary to describe the version of a given discipline.

- Configurations are proposed as a conceptual framework to integrate versions of different disciplines to describe the overall project design. Previous research efforts have not addressed versioning in a distributed environment, and have not presented frameworks similar to the one proposed.

- The model presented in this report considers an instance-instance inheritance scheme. The current model only explores inheritance along the descendant-of relationship in the version hierarchy. Also, the inheritance scheme presented uses a static binding scheme. This may be extended to include dynamic binding among versions along a derivation history.

- Incremental addition of data to a design version is supported by implementing versions in terms of deltas. Work done in the software engineering domain needs to be adapted to design data management context.

- The foundations for storing and managing changes using deltas is the detection of changes due to the execution of an application transaction. This issue has also not been researched among the papers surveyed.

- Version states are distinguished based on properties associated with their definitions. Some attempts to distinguish critical versions have been proposed [21][7]. However, the current work explicates version as well as configuration states that form the foundations of the logical models proposed.
Chapter 3

Versioning Scheme to Support Application Transactions

In this chapter, a model for versions is presented that supports design application transactions in a given discipline. A design transaction model is presented in Section 1. This model is based on a check out / check in protocol that does not permit more than one transaction to access a shared data item at a given time. Section 2 presents a versioning scheme that supports a design transaction. In this model, version states are distinguished by the ability of a transaction to modify its contents. Thus, a design transaction can only check in data to a version whose contents can be modified. The committing of a transaction may, thus, result in the execution of operators on version states. The dependencies between a transaction commit and version states are described in the last section.

3.1 Application Transaction Model

Conventional transactions [14] have the following properties: Atomicity, Consistency, Isolation and Durability (ACID). However, design transactions may not unconditionally support these properties [1]:

- **Design transactions are non-atomic**: Design transactions are typically long duration transactions. A system crash during a design transaction should lead to recovery of as much work as possible, even if it results in an inconsistent intermediate state. Restarting an aborted transaction from the start is not an acceptable solution as a fair amount of effort may be lost.

- **Design transactions need not be consistent**: Design transactions need to support user interactions which are typically nondeterministic in nature. This implies that a transaction cannot be assumed to be a consistent unit. Special validation programs may need to be executed to determine the consistency of the database after the execution of a design transaction.

- **Design transactions need not be isolated**: Cooperation between design transactions has implications on concurrency control mechanisms. Lower degrees of consistency are typically acceptable as design transactions operate in co-operative environment. Additionally, it is unreasonable for a transaction to wait for long durations of time for another transaction to terminate. Also, deadlocks are usually intolerable. In advanced models, different levels of co-ordination among transactions may be desirable, with increasing levels of complexity.
CHAPTER 3. VERSIONING SCHEME TO SUPPORT APPLICATION TRANSACTIONS

- Design transactions desire durability. The effects of a design transaction must be maintained on a non-volatile storage medium. This would facilitate a REDO or UNDO operation of a committed design transaction.

A check out / check in protocol is proposed for the interaction of the transaction with the database. The application transaction checks out all the data that it needs at the beginning of the transaction and checks in data at the end of the transaction.

A transaction that has been initialized is uniquely identified by a system generated transaction identifier (TRID). The outcome of a transaction could either be committed or aborted. The Null state specifies the state when the transaction is no longer linked to a TRID (either before the start or after the completion of the transaction). The check out / check in protocol specifies the following additional states:

- **Active Transaction**: A transaction that has checked out the data that it requires into its workspace is defined as Active.

- **Executed Transaction**: A transaction that has completed its specified design tasks and stored its results in its workspace is in the Executed State.

- **Reserved Transaction**: A transaction that has completely checked the manipulated data back into the database is in the Reserved state.

The transaction model is described in Figure 3.1. The execution of transaction operators (represented by directed bold links) cause the transaction to transition from one state to another. The Begin-Transaction operator initializes a transaction instance and assigns it a TRID. The transaction may
CHAPTER 3. VERSIONING SCHEME TO SUPPORT APPLICATION TRANSACTIONS

Figure 3.2: Model of Versioning to Support Design Transactions

check out from the database all the data that it requires in the check out phase. The Execute-Transaction operator abstracts the execution of design tasks by a transaction process. The design transaction is essentially decoupled from the database during this phase. On the completion of the tasks, the transaction may check in data from the workspace back to the database. The commit operation, essentially releases the locks held on the version by the transaction. A committed transaction cannot be aborted. Aborting the transaction while it is in the reserved state involves an UNDO operation. Recovery from failures during an intermediate phase is achieved using the REDO log.

3.2 Version Model to Support Transactions

The model for the versions to support the data needs of a design transaction is presented in Figure 3.2. The version states are distinguished on the ability of a design transaction to modify its contents. The following states are distinguished:

- **Active Version**: A version whose contents are currently manipulated by an application transaction is defined to be Active. At a given time, there exists at most one version in the Active state in a version hierarchy.

- **Suspended Version**: A version whose contents can potentially be altered by an application transaction is a Suspended Version. To alter the contents of a Suspended Version, it needs to be first transitioned to the Active State.
• **Declared Version:** A version whose contents can only be accessed but not altered by an application transaction is defined to be in the **Declared state**.

• **Removed Version:** A version which used to exist and has since been removed is in the **Removed state**.

Operators are defined to transition a version from one state to another and are represented in the figure by directed bold links. The **Declare-Version** operator defines a version initially in the active state to be in the declared state. Similarly, the operator **Suspend-Version** transitions an active version to the suspended state. **Activate-Version** defines the suspended version on which it operates as active. If a version were defined as active prior to the execution of the operator, it is defined as suspended. The operator **Remove-Version** removes the version specified. A new version may be created by the **Create-Version** operator and may be defined to be active. The version that was active prior to the execution of this operator is now defined as suspended.

Operators that link two version states are represented in the figure by directed dashed links. The operator **Derive-Version** links a newly defined active version as an immediate descendent of a specified declared version. Execution of the operator copies the contents of the parent version into the newly specified child version. An execution sequence of the above operators results in the generation of a hierarchy of version states. This hierarchy specifies the **version derivation history**. Interior nodes in this tree represent declared version states. The contents of the versions specified by the leaf nodes can be modified. Thus, at most one of the leaf nodes represents the version that is active at that time, while the remaining leaf nodes represent suspended version states.

### 3.2.1 Operators on the Data Contents of Versions

Operators are defined to alter the contents of an Active Version. They are essentially operators to **Add** data items, to **Discard** data items, and to **Modify** values assigned to data items. Before altering the contents of a suspended version it first needs to be activated. The contents of a declared version state cannot be altered.

In addition, operators are defined to access the contents of any existing version. The **Read-Data** operator queries values assigned to particular data items in the specified version. The operator **Describe-Version** outputs the contents of the queried version. Retracing of value assignments to data items made in different versions (from a specified version to its ancestors) is performed by the **Trace-Data** operator. The operator **Compute-Deltas** returns the minimal set of changes that need to be made to the contents of the ancestor version to obtain the contents of the descendant version. In other words it computes the **deltas** between two versions, where **deltas** are expressed as a sequence of operations that may be executed on the contents of the ancestor version to obtain the contents of the descendant version.

### 3.3 Dependencies Between Check In and Version State Operations

A transaction can check out data from any existing version. Checking out a data item places an exclusive lock on the versions of the entities accessed. The granularity of the locking scheme is the level of versions. Hence, transactions that need to access the same entity must either wait until the existing locks on the version are released, or they may access other versions of the same entity.
Thus, this scheme is **pessimistic** in the sense that it does not allow access conflicts to occur on the same version, but it is **optimistic** in that it allows multiple parallel versions of the same object to occur even if these versions are conflicting.

On its completion, the transaction attempts to check in data to the version which it initially accessed. However, a transaction can only modify the contents of a version that is defined to be active. Thus, depending on the state of the accessed version at the time of check in, one of the following version state transition operations, summarized in Figure 3.3, may result (if the version were active then there would be no state transition operation):

- If the accessed version was *suspended*, then it may be *activated* before checking in data.
- If the accessed version was *declared*, then a new active version may be created and linked as a descendant of the accessed version by the execution of the *derive-version* operator.
- If the application transaction had not accessed any version (initial design situation), then a new active version may be created by the *create-version* operator.
Chapter 4

Versions to Support Configurations

In this chapter, configurations are proposed as a framework to integrate designs from different disciplines to describe the overall design of the project. A design in a particular discipline is specified by a version state. Interactions exist among designs from different disciplines, as they reference the same geometric space. These interactions are expressed as constraints. Thus, a configuration is formally specified as a set of versions (one from each discipline), and a set of applicable constraints. In addition to specifying interactions among designs, the constraints may also represent restrictions on the contents of individual versions.

In the first section, a model for configurations is proposed. Additional properties need to be specified for a version state for it to be included in a configuration. The version model presented in the previous chapter is enhanced to support configurations and is presented in Section 2. Preconditions on version operators due to configuration inclusion dependencies are presented in the last section.

4.1 Configuration Model

As mentioned earlier, a configuration is defined by a set of component versions and a relevant constraints. It is uniquely identified by a system generated $\text{Config Id}$. A configuration definition may be linked to a base configuration against which it may evaluated. Each version of the base configuration is an ancestor of the corresponding version of the specified configuration. Formally, a configuration definition $CF_k$ is specified as:

$$CF_k = < V_{il} \ (i = 1, ..., n), CS_k, CF_{Base} >$$

where,

$k =$ identifier of the configuration definition
$V_{il} =$ version number $l$ from discipline $i$
$n =$ number of disciplines
$CS_k =$ set of constraints associated with the configuration
$CS_k = \{ cs_j \ | \ j = 1, ..., m \}$
$cs_j =$ $j^{th}$ constraint in the constraint database that is an element of $CS_k$
$m =$ maximum number of constraints included
CHAPTER 4. VERSIONS TO SUPPORT CONFIGURATIONS

$CF_{Base} =$ Base configuration against which the current configuration can be evaluated. For a configuration defined in isolation, this term may have a NULL value assigned.

A configuration is defined by a designer using versions belonging to other disciplines, as well as, a version from his own discipline. A configuration definition references the discipline identifier of the creator of the configuration definition, as well as, the set of constraint violations that may result from the evaluation of the constraint set $CS_k$ using data from the component versions $V_{it}$.

Two configuration states are specified:

- **Defined Configuration**: A configuration that has been specified by a set of component versions and an associated set of constraints.

- **Eliminated Configuration**: A configuration that was previously defined and has since been eliminated.

The model for configurations is presented in Figure 4.1. The state transition operators are represented by directed bold links in the figure. The **Define-Configuration** operator specifies a new configuration definition. A configuration may be eliminated by the **Eliminate-Configuration** operator. The operator **Derive-Configuration** links a newly defined configuration to a base configuration definition and is represented by a directed dashed link.
4.1.1 Operators on the Contents of a Configuration

A configuration is a static structure. Thus, operators are defined that can only access the data referenced by the definition. The operator **Read-Version-Data** queries the contents of the component versions, while the operator **Show-Configuration** outputs the values assigned to all the data items and constraints referenced by the configuration definition. The operator **Characterize-Config-Deltas** characterizes the differences between two configurations in terms of the differences between the contents of corresponding component versions (version deltas), and the differences between the constraint sets associated with both configurations (constraint deltas). The execution of the **Evaluate-Configuration** operator invokes the constraint checking mechanism. The constraint checker returns any resulting constraint violations. These are then linked to the evaluated configuration definition. The violations associated with the evaluated configuration as well as those associated with the base configuration are listed by the execution of the **Report-PC-Violations** operator.

4.2 Version Model to Support Configurations

Restrictions are specified on versions included in a configuration definition. These restrictions are specified as follows:

- A version *cannot be removed* while is a component of an existing configuration. In other words, a version is guaranteed to exist as long as it is a component of any configuration definition.
- The contents of versions from other disciplines used by a designer to specify a configuration must be accessible to the designer.

In order to support configurations, two properties are defined for version states: **Version Status**, and **Version Access Privileges**. Version status describes the protection status of a version state. Version access privileges granted on a version define the ability of users in other disciplines to access its contents. Version states are distinguished on the properties of the version state. A version that cannot be removed is defined as **Frozen**. In other words, a frozen version is guaranteed to exist. A version state whose contents can be accessed by other users is defined as **Published**. The specification of states of component versions in a configuration definition is summarized in Table 4.1. The component versions of users other than the creator of the configuration definition need to be published while the version corresponding to the creator need only be frozen.

The logical model of versions to support configurations is presented in Figure 4.2. The **Freeze-Version** operator guarantees that a declared version cannot be removed. This guarantee is undone by the **Thaw-Version** operator. An active or suspended version cannot be frozen, as guaranteeing the existence of a version implicitly guarantees the maintenance of the contents of that version (i.e., the contents of a frozen version cannot be altered). The **Publish-Version** operator grants other users read access privileges to the values assigned to data items contained in that version. Previously granted access privileges may be revoked by the **Suppress-Version** operator. Dependencies that result from other consultants accessing the contents of a published version require that the accessed version must be protected from being removed. Thus, a version must be **frozen** before it can be **published**.
Figure 4.2: Model of Versions to support Configurations
4.3 Configuration Inclusion Dependencies

Preconditions are specified on the execution of the following version state operators. These result from dependencies due to including a version in a configuration definition:

- **Thaw-Version**: A version cannot be thawed while it is included in a configuration definition.
- **Suppress-Version**: A version cannot be suppressed while it is included in a configuration definition defined by any other user.
Table 4.1: Specification of Version States due to Inclusion Dependencies

<table>
<thead>
<tr>
<th>Configuration Type</th>
<th>Version Properties Other Users</th>
<th></th>
<th></th>
<th>Version Properties Owner</th>
<th></th>
<th></th>
<th></th>
<th>Owner Version State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Protection Status</td>
<td>Visibility to Owner</td>
<td>Others Version State</td>
<td>Protection Status</td>
<td>Visibility to Others</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>defined</td>
<td>freeze</td>
<td>yes</td>
<td>published</td>
<td>freeze</td>
<td>no</td>
<td></td>
<td>frozen</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5

Versioning to Support Collaborative Design

In this chapter, the semantics of configurations are extended to support cooperation among designers. The first section presents a model of configurations to support collaboration. The versioning scheme is enhanced to support the additional configuration states and is presented in the next section. The final section specifies the dependencies that result from including a version in one of the extended configuration definitions.

5.1 Configuration Model to Support Collaborative Engineering

To collaborate with each other, designers may wish to perform the following tasks:

- A designer may wish to modify a design so that it integrates better with the overall project design. In order to do so, a designer must be able to evaluate the design from the perspective of the overall project.

- The design team may wish to evaluate the project design to check for inconsistencies due to interactions among individual designs. Thus, the entire team must be able to see and evaluate a project design.

- A design team may wish to show consultants outside the team the project design. In addition, an individual designer or the entire team may maintain designs for future reference. This requires the maintenance of the project design and providing access to it.

To support these facilities, two additional properties are defined for configuration states: Configuration Status, and Configuration Access Privileges. The former describes the protection status of a configuration definition. For example, a configuration may be explicitly protected from being eliminated. The latter specifies privileges that a designer may grant other consultants on a configuration defined by him/her. The following configuration states are defined based on the properties of the configuration definition:
• **Intermediate**: A configuration definition that is guaranteed to exist, i.e., it cannot be eliminated.

• **Recorded**: A configuration definition that is guaranteed not to be eliminated.

• **Accessible**: A configuration that has access privileges granted to one or more users other than the owner.

• **Landmark**: A configuration that is guaranteed from being eliminated and has access privileges granted to one or more users other than the owner.

The extended model of configurations to support collaboration is presented in Figure 5.1. The **Protect-Configuration** operator guarantees that a configuration definition cannot be eliminated unless it is explicitly **Unprotected**. The **Stamp-Configuration** operator guarantees that the configuration is guaranteed to exist and cannot be unprotected. Access privileges (read access or notify or both) on a configuration may be granted to one or more users by the **Grant-Access** operator. Previously granted access privileges may be revoked by executing the **Revoke-Access** operator. Dependencies that may result from other users accessing a configuration, require that access privileges may only be granted on configurations that are guaranteed to exist (Intermediate or Recorded configurations). Additionally, as a user may only be notified of violations that result on the evaluation of a configuration that may be read accessible, granting notify privileges to a users implies read access privileges.

### 5.2 Version Model to Support Collaboration

A version that is included in a configuration definition that is guaranteed to remain protected (**Recorded** or **Landmark** configuration definitions), must be guaranteed to remain frozen, i.e., it must be prevented from being thawed. The restrictions on the properties of version states included in a configuration definition is summarized in Table 5.1. This motivates the definition of the following additional version states:

• **Archived Version**: A version state that is guaranteed to exist and cannot be thawed.

• **Persistent Version**: An archived version state whose contents are accessible to other users.

The model of versions to support collaboration is presented in Figure 5.2. The **Archive-Version** operator guarantees that the version remains frozen and cannot be thawed.

### 5.3 Configuration Inclusion Dependencies

The state of the configuration places restrictions on its component versions. Thus, the execution of an operator on a configuration state may result in changes in the changes in the restrictions specified on its component versions. This may result in the execution of operators on the states of the component versions. The execution of the following configuration state operators may result in the execution of operators on its component versions:

• The execution of the **Stamp-Configuration** operator archives component versions if they are not already in the archived state.
Figure 5.1: Model of Configurations to support Collaborative Design
Figure 5.2: Model of Versions to support Collaborative Design
CHAPTER 5. VERSIONING TO SUPPORT COLLABORATIVE DESIGN

- The execution of the Publish-Configuration operator publishes the version associated with the owner if it is not already in the published state.

In addition, preconditions may be specified on the execution of operators on versions included in a configuration definition. Thus, a published version may not be suppressed if it is either included in a configuration specified by other users or is included in a configuration defined by the owner of the version, that has access privileges granted to other designers.
Table 5.1: Specification of Version States due to Inclusion Dependencies (Extended Configuration States)

<table>
<thead>
<tr>
<th>Configuration Type</th>
<th>Version Properties Other Users</th>
<th>Version Properties Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Protection Status</td>
<td>Visibility to Owner</td>
</tr>
<tr>
<td>intermediate</td>
<td>freeze</td>
<td>yes</td>
</tr>
<tr>
<td>recorded</td>
<td>archive</td>
<td>yes</td>
</tr>
<tr>
<td>accessible</td>
<td>freeze</td>
<td>yes</td>
</tr>
<tr>
<td>landmark</td>
<td>archive</td>
<td>yes</td>
</tr>
</tbody>
</table>
Chapter 6

Application Example

In this chapter, a simple application example is presented to illustrate the model of versions and configurations proposed in this report. The scenario considered for this example is presented in Figure 6.1. Data associated with the contracting and structural engineering disciplines are considered in this example. A single entity is considered from each of the two disciplines: (i) beam entity (structural database), and (ii) crane entity (contractor database). In an engineering context, the beams considered correspond to the beams at the top level of a two story frame. The cranes considered are the ones available to place the beams at their appropriate position. A restriction due to interactions between the two entities is specified as the following constraint:

For each beam there exists at least one crane for that floor level whose capacity is greater or equal to the weight of the beam.

The schemes for the two entities are also presented in the figure. Section 1 develops a version hierarchy for the beam entity to illustrate the model of versions. In section 2, data corresponding to the beam and crane entities from two different disciplines are integrated to illustrate a specific project function. Thus, a configuration is defined that is composed of versions of the beam and crane entities.

6.1 Illustration of the Versioning Scheme

An initial state of the version hierarchy of the beam entity is presented in Figure 6.2. In the figure, the contents of each version are specified according to a scheme. The initial states of the versions are also specified in the figure. The root version (B1) was initially defined by the Create-Version operator. The root version is defined as Declared. Therefore, its contents cannot be altered. A sequence of operators to be executed on the initial hierarchy is presented in Table 6.1. The operators have been selected such that their preconditions are satisfied. The table specifies the initial and final states of the versions affected by the operators. For example, execution of the Suspend-Version operator transitions version B1A1 from the Active state to the Suspended state. The state transitions due to the remaining operators follows in a similar fashion. The Derive-Version operator modifies the version hierarchy. In this example, the execution of the operator defines version B3 and links it as an immediate descendant of version B2. The contents of the parent version (version B2) are copied into the child version (version B3). Additionally, the newly created version is specified
Beam(Beam_id, Span, Beam_wt, Floor_lvl)  
Crane(Crane_id, Crane_capacity, Floor_lvl)

Figure 6.1: Application Example Scenario

<table>
<thead>
<tr>
<th>Operations</th>
<th>Version Affected</th>
<th>Initial State</th>
<th>Final State</th>
<th>Version Hierarchy Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suspend (B1A1)</td>
<td>B1A1</td>
<td>active</td>
<td>suspended</td>
<td>none</td>
</tr>
<tr>
<td>Activate (B2)</td>
<td>B2</td>
<td>suspended</td>
<td>active</td>
<td>none</td>
</tr>
<tr>
<td>Declare (B2)</td>
<td>B2</td>
<td>active</td>
<td>declared</td>
<td>none</td>
</tr>
<tr>
<td>Derive (B3, B2)</td>
<td>B3</td>
<td>not existent</td>
<td>active</td>
<td>child of B2</td>
</tr>
</tbody>
</table>
Scheme: Beam (Beam_id, Span, Beam_wt, Floor_lvl)

Create-Version

2

< 2, d,16t, 2 >

Declared (B 1)

Derive-Version
Data Operations

5 6 7

Suspended (B 2)
< 5, d/3, 4.5t, 2 >
< 6, d/3, 4.5t, 2 >
< 7, d/3, 4.5t, 2 >

Derive-Version
Data Operations

3 4

Active (B 1A1)
< 3, d/2, 7.5t, 2 >
< 4, d/2, 7.5t, 2 >

Figure 6.2: Initial Version Hierarchy (Beam Entity)
to be *Active*. Table 6.2 presents a set of changes that are checked in to the version hierarchy by an application transaction. As mentioned earlier, an application transaction can only check in data to the version specified as Active. At this point, version *B3* is in the *Active* State. Thus, these changes modify its contents. The final state of the version hierarchy of the beam entity is presented in Figure 6.3. The attribute values of Beams 5 and 6 in version *B3* are due to the changes checked in by an application transaction. On the other hand, the specification of Beam 7 is inherited from version *B2*.

The execution of the *Compute-Deltas* operator is illustrated in Figure 6.4. This operator computes the changes between two versions, where one is an ancestor of the other. These changes are specified as a minimal set of data operators that need to be executed on the contents of the ancestor version to obtain the contents of the descendant version. The changes (deltas) between versions *B1* and *B2* and between versions *B2* and *B3* are computed in the figure. In addition, the changes between versions *B1* and *B3* are also computed.

### 6.2 Illustration of the Configuration Model

This section integrates versions of the beam and crane entities to specify configuration definitions. The constraint presented earlier in this chapter is also included in the configuration definition. A version hierarchy of the crane entity is presented in Figure 6.5. Two configurations definitions are described in Figure 6.6. The operator *Characterize-Config-Deltas* presents the deltas between each of the corresponding component versions and between the set of constraints associated with the configuration definitions. A precondition for the execution of this operator is that each of the versions in one of the configurations must be an ancestor (or descendant) of the corresponding version in the other configuration.

The specification of the states of the component versions due to the definition of the state of a configuration is illustrated in Table 6.3. In this example, a designer from the structural engineering discipline defined Configuration 2. For example, if Configuration 2 is in the Intermediate state, then the version *B3* may be *Frozen*, while the Version *CIA2* needs to be *Published*.

Although the example presented in this chapter is not very realistic, it illustrates the conceptual models presented in this report.
Table 6.2: Operations on Version Contents (Application Example)

<table>
<thead>
<tr>
<th>Beam-id</th>
<th>span</th>
<th>beam-weight</th>
<th>floor-level</th>
<th>op-desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>d/2</td>
<td>7.5 t</td>
<td>2</td>
<td>replace</td>
</tr>
<tr>
<td>6</td>
<td>d/6</td>
<td>3.0 t</td>
<td>2</td>
<td>replace</td>
</tr>
</tbody>
</table>

Active Version: B3

Scheme: Beam(Beam-id, span, beam-weight, floor-level)

Table 6.3: Specification of Version States (Application Example)

<table>
<thead>
<tr>
<th>Configuration State (Configuration 2)</th>
<th>Version State (Version B3)</th>
<th>Version State (Version C1A2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>intermediate</td>
<td>frozen</td>
<td>published</td>
</tr>
<tr>
<td>recorded</td>
<td>archived</td>
<td>persistent</td>
</tr>
<tr>
<td>accessible</td>
<td>published</td>
<td>published</td>
</tr>
<tr>
<td>landmark</td>
<td>persistent</td>
<td>persistent</td>
</tr>
</tbody>
</table>

Structural Database: Beam Entity

Contractor Database: Crane Entity

Configuration Creator: Structural Engineer
Figure 6.3: Final Version Hierarchy (Beam Entity)
Figure 6.4: Computing Deltas between Versions (Beam Entity)
Scheme: Crane (Crane_id, Crane_capacity, Floor_lvl)

Create-Version

< 1, 10t, 2 >
< 2, 10t, 2 >

Declared
(C 1)

Derive-Version
Data Operations

< 1, 10t, 2 >
< 3, 8t, 2 >

Suspended
(C 2)

Derive-Version
Data Operations

< 3, 9t, 2 >
< 4, 6t, 2 >

Declared
(C 1A1)

Derive-Version
Data Operations

< 4, 6t, 2 >
< 5, 5t, 2 >

Active
(C 1A2)

Figure 6.5: Final Version Hierarchy (Crane Entity)
Beam Version Hierarchy:
Beam (Beam_id, Span, Beam_weight, Floor_level)

Crane Version Hierarchy:
Crane (Crane_id, Crane_capacity, Floor_level)

Figure 6.6: Configuration Definitions (Application Example)
Chapter 7

Summary and Conclusions

In this report, a model is proposed to support the data management needs for collaborative design. Designs in a single discipline are managed by a versioning scheme. Restrictions among the designs in different disciplines are represented by constraints. Configurations are defined to integrate the designs in different disciplines to represent the overall project design. Thus, a configuration is defined by a set of versions and an associated set of constraints. Additional semantics are provided to the configuration states to support cooperation among the designers.

The satisfiability of the model as a solution to the data management needs for collaborative design is discussed in Section 1. The current status of the research effort is presented in Section 2. The last section identifies future research work to enhance the proposed model.

7.1 Evaluation of the Proposed Model

The model presented supports the data management needs of design applications in a given discipline. In addition, it supports the integration of individual designs. Thus, the satisfiability of the model must be addressed both for the design in a single discipline, as well as, for collaboration among multiple disciplines. The first subsection evaluates the model as a solution to the data management needs of a design application in a single discipline. The second subsection focuses on the facilities provided by the model to support the integration of designs in multiple disciplines.

7.1.1 Support for Data Management Needs of a Design Application in a Single Discipline

Design is an evolutionary process. In this model, versions represent specific design states. The model permits the incremental addition of data to the Active version state. Execution of the Derive Version operator links versions as descendants of previously declared versions to generate a version derivation history, which is implemented as a tree structure. The maintenance of the derivation history permits a designer to redesign from an earlier version (previous design state). Also, branching allows multiple alternatives to be generated. The notion of Suspended versions enables design alternatives to be independently developed into sub-hierarchies. Archiving a version guarantees that the version cannot be removed. Thus, an archived version can be maintained for future records. The operator Compute-Deltas determines the changes between two versions, where one is an ancestor of the other.
Although this model does not address the evaluation of constraints, it supports a constraint checker. Constraints local to a discipline represent restrictions within the discipline.

### 7.1.2 Support for Collaboration and Integration of Designs in Multiple Disciplines

Configurations provide a framework to integrate designs from different disciplines and to detect inconsistencies among them. A *Defined Configuration* links versions at different database sites and a relevant set of constraints that specify both restrictions on individual component versions, as well as, interactions between different component versions. Additional mechanisms are supported to facilitate collaboration:

- The *Intermediate* configuration state enables a designer to privately evaluate his design, with respect to designs in other disciplines. Additionally, only he is notified of any violations that may result on evaluating the configuration.
- The *Accessible* configuration state facilitates the entire design team to evaluate a specific project design.
- *Landmark* and *Recorded* configuration states maintain important project designs for both individual designer, as well as, design team records.

The notions of *Frozen* and *Published* version states are defined to support configuration definitions.

### 7.2 Current Status of Research

The development of the version and configuration model can be subdivided into the following tasks: (i) definition of a versioning scheme at each database site in terms of versions of individual entities, (ii) definition of configurations in terms of component site versions, and associated constraint set, (iii) support for incremental evaluation of configurations, (iv) characterization of violations in terms of the constraints that were violated and the inconsistent referenced data, and (v) detection of changes caused by the execution of an application transaction.

The implementation of the model has been divided into two phases. The objective of the first phase is to test the validity of the proposed model. The simple example presented in Chapter 6 is used to validate the correctness of the model. Phase 2 aims to test the scalability of the configuration model to test realistic engineering problems. The testing schemes for this phase are still under investigation.

### 7.3 Future Work

The following extensions are proposed for the current work:

- Extension of the *is-derived-from* link in the version derivation hierarchy to support dynamic inheritance among versions. In this work, two types of *is-derived-from* links have been identified: (i) *copy* mode, and (ii) *reference* mode. In the copy mode link, the contents of the parent version are copied into the child version at the time of its derivation. This supports a
static inheritance of data by a version from its parent. On the other hand, the reference mode link resembles the traditional OOP paradigm. Currently, only the copy mode link has been explored. The model may be extended to support dynamic inheritance schemes by utilizing reference mode links.

- Detection of changes made by an application transaction. As mentioned earlier, the foundations of the proposed model is the management of design changes. Thus, a change detection module must be developed as a wrap around applications to permit them to interact with the model.

- Development of a recursive configuration model. The current model only supports two levels of assemblies of entities - the version and the configuration level. A recursive configuration model would enable the management of multiple levels of assemblies of entities as a configuration could be composed of other configurations or versions.
Bibliography


