CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

AN INTEGRATED FRAMEWORK FOR DESIGN STANDARDS PROCESSING

by

Nobuyoshi Yabuki
Kincho H. Law

TECHNICAL REPORT
Number 67

June, 1992

Stanford University

SUMMARY
CIFE TECHNICAL REPORT #67

Title: An Integrated Framework for Design Standards Processing

Authors: Nobuyoshi Yabuki, Ph.D., P.E., Electric Power Development Co., Ltd., Japan, and
Kincho H. Law, Associate Professor, Department of Civil Engineering.

Date: June 1992

Funding: Fellowship from the Electric Power Development Co., Ltd., Japan,
Fulbright Scholarship from the Japan-United States Educational Commission,
Research Fund from the National Science Foundation, Grant No. IRI-9116646.

1. Abstract:

This report describes an integrated framework for the documentation, representation, and processing of design standards in a Computer Aided Design (CAD) environment. We combine object-oriented and logic programming paradigms to provide a unified Object-Logic model to represent and process design standards. In this model, a designer can check the design for compliance with design provisions as well as perform component design. The framework also includes the storage of background information such as commentaries of the design provisions in a HyperDocument environment. To demonstrate the feasibility and practicality of this framework, a prototype system has been implemented for the American Institute Steel Construction (AISC) Load and Resistance Factor Design (LRFD) specification.

2. Subject:

This report describes the results of a research project that studies the representation, processing, and documentation of design standards. In this report, we reviewed and examined previous and current related work to this research, described an integrated framework for design standards processing in detail, and demonstrated the feasibility of this framework. The key idea of this framework is that the combination of the object-oriented and logic programming paradigms is suitable for representing the organization of the design standard and for representing and processing design provisions both for component design and conformance checking. The other key idea is the importance of storing and utilizing background information both for code writers and design engineers. The framework integrates the model for representation and processing of design standards with the model for the documentation of design standards and their background information and knowledge.

3. Objectives/Benefits:

Design standards play a significant role in the design process to ensure safety, quality, and functionality of civil engineering structures and facilities. Design standards contain a large amount of complex information; thus an engineer would need a great deal of experience to comprehend and use the code correctly and effectively. Conformance checking and designing using standards is a tedious, laborious, and difficult task. The objective of this research is to develop a model that can:

- perform both conformance checking and component design within the same environment,
- represent both the organization and provisions of the design standard effectively,

- check the completeness and consistency of design standards,
- store background documents and knowledge that can be accessed by code writers and engineers, and
- be integrated in a CAD environment.

The model would automate and enhance the design and conformance checking process. In addition, the model would enhance and facilitate the code developing and revising process.

4. Methodology:

The background and status of current research on design standards processing and documentation systems were acquired through literature review. A new model and a methodology were developed by the authors. A prototype system was implemented and tested on a few sample problems.

5. Results:

The results of this research can be summarized as follows:

- A new representation scheme for design standards processing through a combination of object-oriented and logic programming paradigms was developed.
- A methodology to perform both conformance checking and component design within the same design environment was established using the Object-Logic scheme.
- A model for storing and utilizing heterogeneous documents of background information and knowledge was developed and integrated with the design standards processing model.
- A framework for checking completeness, uniqueness, and correctness of design standards was developed.
- A framework for integrating a design standards processing system with other design applications was developed.

A prototype system has been implemented.

6. Research Status:

This research has been completed. The framework developed in this research project can be readily applied to developing design standards processing systems. Nobuyoshi Yabuki is planning to apply this model to a Japanese penstock design standard.

Abstract

This thesis describes an integrated Hyper-Object-Logic model for the documentation, representation, and processing of design standards.

There are two distinct categories of knowledge in a design standard: (i) knowledge of the organization of design objects and (ii) knowledge of the methods used in reasoning about design. The object-oriented paradigm lends itself naturally to representing the organizational aspect of the design standard. The logic programming paradigm, on the other hand, is well suited to implementing the reasoning mechanisms for design and conformance checking. The object-oriented and logic programming paradigms are combined to provide a unified Object-Logic model for the representation of design codes and the processing of design standards. By storing the design provisions in a knowledge base, the model is capable of performing conformance checking and component design, and syntactically analyzing the applicable standards.

Besides the Object-Logic representation, this thesis also addresses the issue of the storage of background information (such as commentaries) related to the design provisions. A HyperDocument model, which is based on the HyperFile structure, is developed for the documentation of design provisions. The HyperDocument model provides an organizational model to store and access heterogeneous documents, including design provisions, their background information and data, and programs that can enhance the process of developing and revising design standards. In addition, the documentation system can serve as a means to provide explanations and background information that are needed to support design tasks.

The Object-Logic and the HyperDocument models are integrated into a unified Hyper-Object-Logic model. To evaluate the feasibility and practicality of this model, a prototype system, HyperLRFD++, has been implemented for parts of the American Institute of Steel Construction (AISC) Load and Resistance Factor Design (LRFD) specification and tested on a few sample problems.

Acknowledgments

This report is reproduced from a doctoral dissertation by Nobuyoshi Yabuki to Stanford University. The doctoral committee comprised of Professors Kincho H. Law (principal advisor), Helmut Krawinkler, Raymond E. Levitt, and William J. Rasdorf (North Carolina State University).

We would like to acknowledge the sponsorship for the graduate study of Nobuyoshi Yabuki at Stanford University by the Electric Power Development Co., Ltd. and the Japan-United States Educational Commission (Fulbright Scholarship). This research is also partially supported by the National Science Foundation, Grant No. IRI-9116646.

Some pictures in the figures of this report were optically scanned and reproduced from the AISC LRFD Manual [AISC 86] and one part of the AISC *Engineering Journal* [Yura 71] with the permission of the publisher, the American Institute of Steel Construction, Inc.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Design Standards play a significant role in the design process to ensure safety, quality, and functionality of civil engineering structures and facilities. Design standards contain a large amount of complex information; thus an engineer would need a great deal of experience to comprehend and use the code correctly and effectively. Checking a design for conformance with applicable design codes (conformance checking) and designing using standards is a tedious, laborious, and difficult task. Misinterpreting or overlooking provisions of a design code could have serious consequences.

A design standard can be defined as a document that states the requirements that must be met in order to ensure that an adequate level of performance for an entity is provided [Garrett 86]. Design standards can be divided into three basic types [Fenves 76]:

- performance standards, which state the required performance in a specific way but contain no procedure for evaluating the performance.
- procedural standards, which consist of both statements of required performance and procedures for evaluating the performance.
- prescriptive standards, which strictly state the required dimensions and properties of components in a design.

In structural design, procedural standards are most commonly used, e.g., American Institute of Steel Construction (AISC) specifications [AISC 86] [AISC 89]. This thesis focuses on procedural standards and they are referred to hereafter as design standards or simply standards. In addition, within the context of this thesis, design codes and specifications are treated as synonyms of design standards. A design standard consists of a set of provisions, each of which stipulates some quality, function, or performance of a product or process. Provisions are represented in a form of sentences, tables, charts, or equations. This thesis concerns with the design of structural components in compliance with the provisions.

For more than two decades, many researchers and engineers have attempted to computerize design standards for design application and conformance checking purposes. The recent developments in artificial intelligence and hypertext technologies could have significant impact on the representation of design codes, the processing of design standards, and the integration of design standard processing in an intelligent Computer Aided Design (CAD) environment. This thesis identifies the problems in the current and previous approaches for design standards processing and proposes a model for the representation, processing, and the documentation of design standards.

1.1 Problem Statement

A common approach for incorporating design standards processing into CAD software is to encode the provisions into a computer program using procedural languages. This approach poses the following problems:

- It is difficult to modify the program when the standard is revised because the provisions are "hard-coded" into the program.
- The developers of the design software tend to be computer programmers or junior engineers who may not be familiar with the design provisions. They may misinterpret the code provisions, making the program invalid.
- "Hard-coded" programs often aim for either conformance checking or automated designing, but it is rather difficult to implement both applications in a unified manner.

Various models have been developed for representation and processing of design standards. However, current models for standards processing as well as "hard-coded" systems have no direct mapping between the provisions of design standards and the program code representing the provisions. It is difficult to check the consistency between the program code and the provisions.

Design code development is often a long and tedious process with many discussions, writings, and rewritings. Although every effort is made to ensure the correctness of the specification, it remains difficult to ensure that the specification is free of mistakes, inconsistencies, and incompleteness. The methods currently used for checking standards are rather trivial and error prone. Code issuing authorities distribute "draft" standards to researchers and practitioners and depend on the human experts to detect possible conflicts and errors. The effort required to identify inconsistencies is enormous if the standard is complicated and voluminous. It is desirable to have a systematic mechanism that can check inconsistencies in a design standard represented in a certain computerized format.

During the code development process, many necessary documents and data are collected. Collecting these background data and knowledge is time-consuming and expensive. Unfortunately, the documents collected are often not shared by other researchers and engineers. When the standard is revised, extra efforts are required to re-collect many of the documents. It is thus desirable to develop a document storage system that can store and access a large amount of various, heterogeneous documents containing text, charts, drawings, pictures, video, and audio.

Due to the growing complexity of civil engineering projects and the increasing time pressure on design, engineers often lack the time to study the fundamentals and background of new or revised design standards. In practice, designers, especially junior designers, tend to plug numbers in design equations without fully understanding the meaning and implications of the provisions they use. Consequently, designers may misinterpret and misunderstand the implications of a provision. Designers cannot achieve an economical and safe design because they may not be able to distinguish the important parameters from less important ones in terms of economy and safety. It is desirable to include explanations and background information for easy access upon request.

In knowledge engineering, there are currently two approaches for providing explanations to the "why" questions:

- showing explanations for anticipated questions (canned text approach), and
- paraphrasing the rules or the methods encoded in the program into English.

The drawback of the canned text approach is that the developer has to anticipate all the questions the users may ask. The limitation of the paraphrasing approach is that it is weak in justifying what the program does or did [Swartout 83]. We need a more flexible explanation facility that can explain justification. The knowledge required to provide these justifications is often available to produce the design standard but is seldom recorded as part of the program or standard. Thus, the document storage system, as pointed out earlier, could be a useful means to provide explanations to the user. That is we need appropriate linkages and integration between the design or checking software and the document system.

Current conformance checking programs that are available in the industry are mostly stand-alone type. They are usually not integrated with CAD data models, engineering databases, and Computer Aided Design and Drafting (CADD) systems. The user of such a conformance checking system must supply to the system all the necessary input data or design plans and documents by typing in the information from a keyboard. The issue of integrating these heterogeneous application programs is also addressed in this thesis.

1.2 Objectives

The overall goal of this thesis is to develop an integrated framework for design standards representation, processing, and documentation in an intelligent CAD environment. The objective of this research is to develop a model that can:

- perform both conformance checking and component design within the same environment,
- represent both the organization and provisions of the design standard effectively,
- check the completeness and consistency of design standards,
- store background documents and knowledge that can be accessed by code writers and engineers, and
- be integrated in a CAD environment.

To demonstrate the feasibility and practicality of the model, a prototype system is developed in this study.

1.3 Scope of Study

The scope of the design process addressed in this thesis is limited to component design and conformance checking. System design and analysis are not included in the scope of this thesis. However, with appropriate interfaces, the developed model can be integrated with system design and analysis tools.

The prototype system is developed solely for the purpose of demonstration and evaluation of the proposed model for the representation, processing, and documentation of design standards. The examples used to illustrate the model are limited to steel structural components such as beams and columns. The design standard used in the prototype development is the AISC Load and Resistance Factor Design (LRFD) specification [AISC 86]. Furthermore, the prototype implementation is limited to compression, flexural, and flexural-compression members of standard wide flange W shape sections. However, the model should be extendable to other type of design standards and component members.

1.4 Organization

This chapter presented the problems and objectives of this thesis. There are seven chapters following this one. The next chapter reviews previous and current work related to this study. Several models and approaches for design standards representation and processing are examined, evaluated, and discussed. Chapter 2 also introduces current hypertext technologies, and assesses their application for design standards representation and documentation.

Chapter 3 presents an overview of a Hyper-Object-Logic model developed in this research for the representation, processing, and the documentation of design standards. In addition, a prototype system, HyperLRFD++, for the AISC LRFD specification [AISC 86] and its capabilities are introduced briefly.

Chapters 4 and 5 describe in detail the methodologies for representing design standards and their background information. The Hyper-Object-Logic model is an integration of two models: an Object-Logic model, which represents design standards for design application and standards analysis, and a HyperDocument model, which stores various documents including design provisions and their background information. Chapter 4 describes a framework for representing the organization and design provisions and for defining design members, employed in the Object-Logic model. Chapter 5 describes the HyperDocument model.

Chapter 6 and 7 describe the procedural modules and applications of the Hyper-Object-Logic model. In Chapter 6, two program modules for conformance checking and component design are described; design examples are presented to demonstrate the capabilities of the HyperLRFD++ system. Chapter 7 describes the methodologies for analyzing design standards and presents examples to illustrate the procedures in ensuring consistency and uniqueness of design standards.

Chapter 8 summarizes this thesis and discusses the limitations of the prototype system and possible future research directions.

Chapter 2
Related Work

This chapter surveys research work related to this thesis. In addition, this chapter reviews the terminology, organization, and operation of knowledge-based system technologies employed for representing and processing design standards. Section 2.1 reviews standards processing models and implementation approaches. Section 2.2 presents the current state of hypertext and HyperFile technologies. Section 2.3 summarizes the discussions in this chapter.

2.1 Standards Processing Models and Implementation Approaches

This section reviews approaches for design standards processing and automated component design using standards. Section 2.1.1 discusses the SASE model, which is implemented using decision tables. Section 2.1.2 reviews design standards processing using production rules. Section 2.1.3 examines the frame-based and object-oriented approaches for the representation of design standards. Section 2.1.4 describes the work on logic-based approach for design standards processing. Section 2.1.5 reviews previous approaches for automated component design based on design specifications.

2.1.1 The SASE Model

The most mature model for standards representation developed so far is the SASE (Standards Analysis, Synthesis, and Expression) model [Fenves 87]. The model uses decision tables for the representation and processing of design standards [Fenves 66]. In the SASE model the basic unit of a standard is a provision, which stipulates some quality of a product or process. Based on their usage in the standard, provisions can be separated into two basic types:

- Requirements — provisions that directly determine compliance with some portion of a standard. Such provisions normally can be characterized as satisfied or violated.
- Determinants — all provisions that are not requirements. Determinants are normally characterized by either numerical or nominal values.

The SASE model consists of four basic components:

1. data items, which represent all the variables in the standard,
2. decision tables, which represent the logic used to determine the values of data items,
3. information networks, which represent the precedence relations among the data items, and
4. an organization system, which represents the organization of the provisions.

These four components are examined in the following subsections.

2.1.1.1 Data Items

Data items represent all the variables in the standard. A data item may be one of four types:

- a numeric quantity such as "F_y" (yield stress);
- a specific value of "satisfied," "violated," or "not applicable";
- a boolean value of "TRUE" or "FALSE"; or
- a member of an enumerated set, such as "compact," "noncompact," or "slender."

A data item is also distinctly classified into:

- a basic data item, which has no ingredients from within the standard to define its value;
- a derived data item, which has both ingredients and dependents to derive its value; and

- a requirement data item, which has only ingredients to derive special value of "satisfied," "violated," or "not applicable."

The dependent of a data item "A" is a data item that uses "A" to compute itself. The ingredient of a data item "A" is a data item that is used to compute the value of "A."

2.1.1.2 Decision Tables

Decision tables represent the logic used to determine the value of data items. Each decision table is responsible for producing a value for one (and only one) data item. A decision table consists of three components:

- conditions — boolean expressions related to the choice of the value for the data item, which may be true (T), false (F), or immaterial (I),
- actions — the possible values for the data item, which are either symbolic expressions which can be evaluated or constants, and
- rules — which prescribe one (and only one) action for a given set of condition values.

There are two kinds of decision tables:

- limited entry decision tables, in which condition values are restricted to T, F, or I.
- extended entry decision tables, in which condition values can be any expressions.

The decision tables used in the SASE Model are restricted to limited entry decision tables. As an example, a provision for computing the design compressive strength $\phi_c P_n$ of a member based on the AISC LRFD specification [AISC 86] is shown in Figure 2-1. A decision table for determining the data item F_{cr} in the provision is shown in Figure 2-2.

2.1.1.3 Information Network

An information network is used to represent the precedence relationships among the data items of the standard. The network is composed of nodes and directed links. Each node represents one data item in the standard. A directed link indicates that the data item of the initial node is a direct ingredient of the data item of the end node. An information network of the requirement for a compression member as described in Figure 2-1 is shown in Figure 2-3. The node farthest to the left (which does not have any successor nodes) such as "Requirement for compression members" in Figure 2-3 is called a terminal data item or a

E2. DESIGN COMPRESSIVE STRENGTH

The design strength of compression members whose elements have width-thickness ratios less than λ_r of Sect. B5.1 is $\phi_c P_n$

$$\phi_c = 0.85$$

$$P_n = A_g F_{cr} \qquad \text{(E2-1)}$$

for $\lambda_c \leq 1.5$

$$F_{cr} = (0.658^{\lambda_c{}^2})F_y \qquad \text{(E2-2)}$$

for $\lambda_c > 1.5$

$$F_{cr} = [0.877/\lambda_c{}^2]F_y \qquad \text{(E2-3)}$$

where

$$\lambda_c = \frac{KL}{r\pi}\sqrt{\frac{F_y}{E}} \qquad \text{(E2-4)}$$

A_g = gross area of member, in.^{2}
F_y = specified yield stress, ksi
E = modulus of elasticity, ksi
K = effective length factor
L = unbraced length of member, in.
r = governing radius of gyration about plane of buckling, in.

For members whose elements do not meet the requirements of Sect. B5.1, see Appendix B5.3.

Figure 2-1 Provision E2 from the AISC LRFD specification [AISC 86]

			Rules	
			R1	R2
Conditions	C1	$\lambda c \leq 1.5$	T	F
Actions	A1	$F_{cr}=\left(0.658^{\lambda_c^2}\right) F_y$	x	
	A2	$F_{cr}=\left[\dfrac{0.877}{\lambda_c^2}\right] F_y$		x

where T : true
F : false
x : Take the left action.

Figure 2-2 Decision Table for Determining F_{cr}

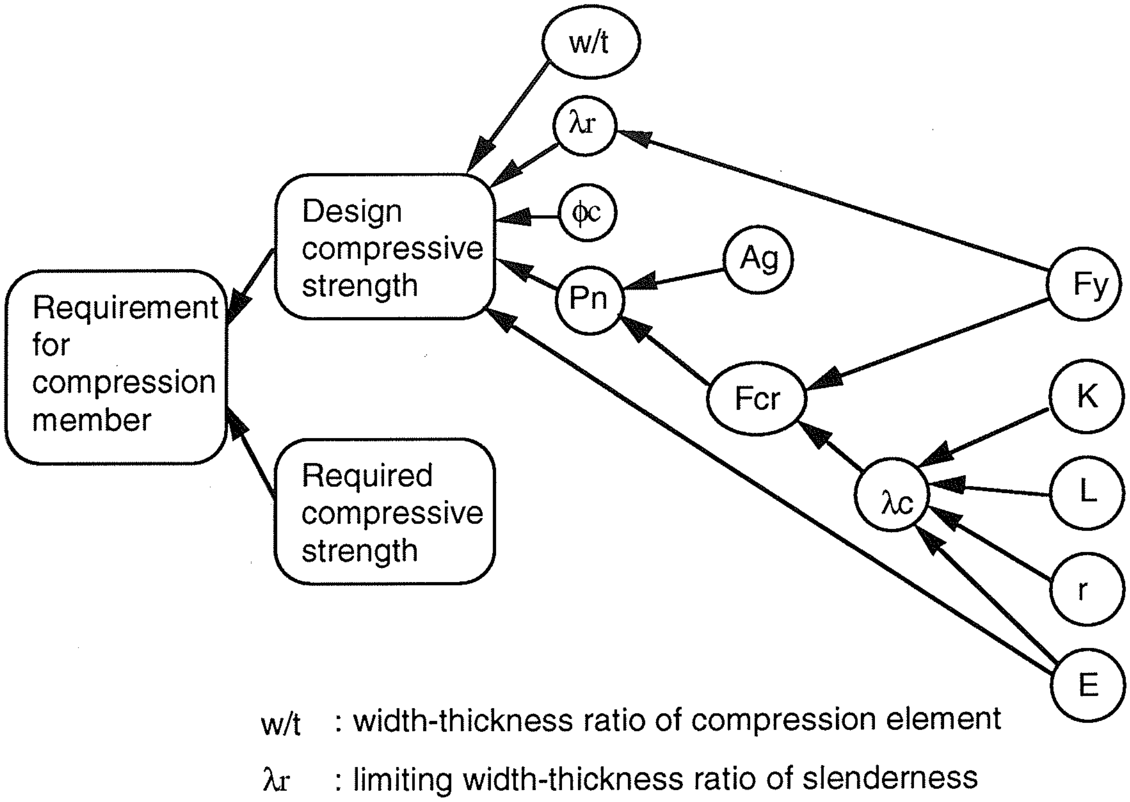

w/t
λr
Design compressive strength
φc
Ag
Fy
Requirement for compression member
Pn
Fcr
K
λc
L
Required compressive strength
r
E
w/t : width-thickness ratio of compression element
λr : limiting width-thickness ratio of slenderness

Figure 2-3 Information Network for the Requirement for Compression Member

requirement. The intermediate node such as "P_n," "A_g," and "F_{cr}" represent derived data items or determinants. The nodes farthest to the right (which do not have any predecessor nodes) such as "F_y," "K," and "L" are the input or basic data items.

2.1.1.4 Organizational System

The user of a standard must be able to identify which provisions of the standard apply for a given design situation. The standard, therefore, needs to be organized in a systematic manner such that individual provisions can be accessed easily. An organizational system can also be used to develop an outline to arrange the provisions and to define the scope of the standard.

A provision of requirement type generally contains two basic components, a subject and a predicate. The provision has the general form:

<subject> <predicate>.

The subject may be a physical entity (e.g., a part of a dam), a process (e.g., design or construction), or a participant in the process (e.g., a designer or contractor); the subject is referred to as THING. The predicate defines the particular quality required of the subject (e.g., strength or stiffness of a structural member or submission of quality assurance document from a manufacturer); the predicate is referred to as REQUIRED QUALITY.

Requirements in design standards can be viewed as expression of behavior limit states. A requirement that represents a single behavior limit state is called a single requirement, which has a single subject and a single predicate. A requirement that can be applied to several behavior limit states is called a multiple requirement, which has multiple subjects or predicates. The advantage of using only single requirements is that each behavior limit state can be explicitly addressed in design. If a provision is a multiple requirement, it can often be decomposed into single requirements.

In the SASE model, the methodology for classification of provisions is based on the faceted classification system developed for library science. The classification consists of several more or less independent areas, called fields and facets. A field can be thought of as a subject area (e.g., a structural component) and a facet can be thought of as a way to classify within a particular field (e.g., material, form, and function). Within the organizational system, there must be at least two independent fields: one for THING and one for REQUIRED QUALITY of the provisions. Each field is further subdivided into facets. Each facet is a hierarchical tree with several levels, and must be a strictly logical tree. That is, each succeeding level, called classifiers, must be a direct subdivision of the parent node in the hierarchical tree and the logical principles of mutually exclusiveness and collectively exhaustiveness must be satisfied at each level [Fenves 87].

As an example, partial facets from two fields, physical entity and limit state, are shown in Figure 2-4 [Harris 81] [Garrett 86]. Appropriate classifiers from each field are associated with each requirement. This association provides a means for accessing the requirements. The set of classifiers associated with a requirement is called an argument list. Table 2-1 shows the example for the requirement identification numbers, their names, and associated

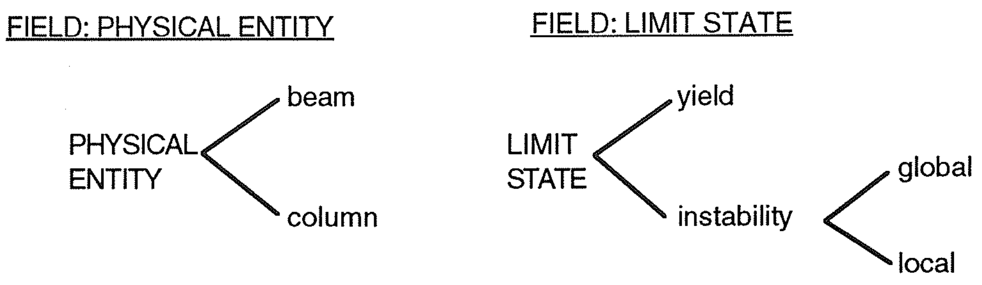


Figure 2-4 Partial Facets of Two Fields [Harris 81] [Garrett 86]

Table 2-1 Requirement Names and Argument Lists [Harris 81]

Number	Requirement Name	Argument List
Req 1	Plastic moment capacity	beam, yield
Req 2	Lateral torsional buckling	beam, instability, global
Req 3	Axial force capacity	column, yield
Req 4	Euler buckling	column, instability, global
Req 5	Local buckling	PHYSICAL ENTITY, instability, local

argument lists. By merging all of the fields together to form one unified tree of classifiers, we can locate all the requirements within that tree to form an outline of the standard. Each leaf node of the tree must have at least one requirement. The outline contains the organization, which is a tabular arrangement of headings, each of which corresponds to a specific classifier, and the pertinent requirements classified under the selected heading. The outline of the standard requirements can be altered by changing the order in which the fields and facets are merged. Two examples of possible outlines for the same set of requirements are shown in Figure 2-5.

(A) First Possible Outline

```
Chapter  Section  Subsection  Requirement Number and Name

1. Yield
     1-1. Beam ————————————   Req 1 : Plastic moment capacity
     1-2. Column ——————————   Req 3 : Axial force capacity
2. Instability
     2-1. Local ———————————   Req 5 : Local buckling
     2-2. Global
          2-2-1. Beam ————    Req 2 : Lateral torsional buckling
          2-2-2. Column ——    Req 4 : Euler buckling
```

(B) Second Possible Outline

```
Chapter  Section  Subsection  Requirement Number and Name

1. Beam
     1-1. Yield ———————————   Req 1 : Plastic moment capacity
     1-2. Instability
          1-2-1. Local ———    Req 5 : Local buckling
          1-2-2. Global ——    Req 2 : Lateral torsional buckling
2. Column
     2-1. Yield ———————————   Req 3 : Axial force capacity
     2-2. Instability
          2-2-1. Local ———    Req 5 : Local buckling
          2-2-2. Global ——    Req 4 : Euler buckling
```

Figure 2-5 Two Possible Outlines [Harris 81] [Garrett 86]

2.1.1.5 Checking Completeness, Uniqueness, and Correctness

Since design standards are the primary governing means for design, the quality of the built environment depends on the quality of the standard. To assure the quality of the design standard, it is essential to check the basic properties of completeness, uniqueness, and correctness at the provision, information network, and organizational levels [Fenves 77]. The requisite qualities at these three levels are identified as follows [Fenves 87]:

1. Individual provisions must be:
   - Unique — the provision must generate one and only one result in any possible condition;
   - Complete — the provision must be applicable to all possible conditions; and

   - Correct — the result of applying the provision must be consistent with the objective of the standard.
2. The information network must be:
   - Connected — each provision must show all the data required for the application of the provision.
   - Acyclic — the data produced by the evaluation of a provision should not be required prior to its evaluation (no loops in logic); and
   - Consistent — uniform logical and technical bases must be provided for comparable provisions.
3. The standards organizational must be:
   - Complete — explicit scope must be provided so that a user knows the subjects and qualities covered by the standard; and
   - Clear — the arrangement and display of provisions should be such that all provisions pertinent to the user's query can be readily found.

For each decision table representing a provision, uniqueness (lack of contradiction and lack of redundancy) and completeness can be checked. The procedures are straightforward. Decision tables must be limited entry type, which each condition must be evaluated to be either true, false, or immaterial. If one combination of conditions is a subset of another, the two rules are redundant. If one combination of conditions indicates two or more exclusive actions, the rules are inconsistent, and if the actions are not exclusive, the rules are ambiguous. If not all combinations of conditions are covered in the decision table, the rules are incomplete. Since combinations of conditions are represented in a tabulated matrix, a computer program for checking the completeness and uniqueness of provisions is to examine the values within the matrix. If the information networks are used to represent the relations of decision tables, the properties of connectedness and acyclicity can be checked graphically. The organization of the standard is represented in a classification system, where the requisite properties can be checked when the code writers make an outline of the standard. Most of these facilities have been implemented in the SASE program [Fenves 87].

2.1.1.6 Discussion on the SASE Model

The first significance of the SASE Model is the separation of the contents of the standard and its computing processor. Contents of the standard are represented in a generic form of decision tables, that not only programmers but also code writers and engineers can write or check. The program for processing decision tables is generic so that modification of contents of the standard represented as decision tables do not affect their processing program. This feature has overcome the problem of the "hard-coding" approach. The second significance lies in the generic organizational system that designers can select applicable requirements and code writer can examine the organization of the standard systematically. However, although it is natural to apply the decision table technique to conformance checking, it is very difficult to apply the technique to generate component design. Only limited efforts for component design have been made, which are described in Section 2.1.5.

2.1.2 Production Rule Approach

A production rule system consists of four basic components:

- Knowledge base,
- Inference engine,
- Working memory, and
- User Interface.

The knowledge base contains a set of rules of the form:

IF [condition] THEN [action].

The inference engine is a control system that interprets rules in the knowledge base, controls the order in which the rules are fired, and resolves conflicts if more than one rule is applicable. Working memory is a set of basic data structures that represent the current state of the system. User interface obtains the input data from users and presents the results to them.

There are two basic reasoning procedures to draw conclusions from the knowledge base:

- backward chaining, and
- forward chaining.

In backward chaining a hypothesis is posed to the working memory. The inference engine confirms the hypothesis by reasoning backward, i.e., from the THEN part [action] to the IF part [condition]. In forward chaining facts are given to the working memory. The inference engine matches the facts with the IF part [condition] of the rules. For the matched rule, the THEN part [conclusion] is added to the working memory as new facts.

2.1.2.1 The Production Rule Approach for Design Standards Processing

With the emergence of expert system technologies, the production rule approach has been proposed by several researchers for representing and processing design standards [Rosenman 85] [Rosenman 86] [Dym 88] [Rasdorf 88] [Kumar 89]. In this approach, the provisions of a standard are represented as production rules instead of decision tables. The production rule approach is a more natural way of representing design standards than decision tables.

As noted by Rasdorf et al. [Rasdorf 88] and Kumar [Kumar 89], the production rule approach has the serious drawback of requiring a very large memory space. This requirement results in a large number of rules that in practice could become unmanageable because one provision may have to be represented as more than one production rule. Even though it is possible to cast the decision table entries as production rules, one is likely to end up with a large number of rules, requiring more memory than the equivalent decision table representation. One approach is to represent production rules as facts to solve the memory problem [Rasdorf 88] [Kumar 89].

2.1.2.2 Checking Anomalies in a Production System

Although no formal mechanisms have been proposed for checking design standards using production rule approach, various methodologies have been developed for checking anomalies in rule-based expert systems. Suwa et al. [Suwa 82] developed a program for verifying knowledge base completeness and consistency (lack of conflicting, redundant, or subsumed rules) in the context of the ONCOCIN system, a rule-based system for clinical oncology. Another program for checking consistency, cyclic rules, and completeness of a generic rule-based expert system has been reported by Nguyen et al. [Nguyen 87]. Cragun et al. developed a decision-table-based processor for checking completeness and consistency in rule-based expert systems [Cragun 87]; in this system, production rules are translated into decision tables, which are checked by the processor. This checking procedure is also employed in the SASE program.

2.1.3 Frame-Based and Object-Oriented Representation of Design Standards

A "frame" is a knowledge representation technique for objects. An object may be a physical object or an abstract concept such as a class of objects or even a theory. A frame is a data structure composed of slots. Slots may have simple values, pointers to other frames, called facets, or procedures that can compute the slot value. The slots of a frame represent the attributes of an object, and slot values represent the specific attribute values of an instance of such an object. The basic inference principle is inheritance. If a frame represents a class of objects (such as building components) and another frame represents a subclass of this class (such as beams), the subclass frame can inherit values from the superclass frame. However, if the subclass has exceptions to the subclass slot values, the subclass can override the superclass slot values.

Object-oriented programming is a relatively new paradigm and has been defined differently by different people. An emerging agreement is that a fully object-oriented programming language would include classes, inheritance, objects, message passing, encapsulation, and polymorphism [Korson 90]. A class is a template that defines the general characteristics of its sub-classes or objects. It contains attributes for storing data and methods (or procedures) for handling messages sent to objects of that class. Classes are hierarchically organized and inheritance is the ability of a class to inherit attributes and methods from its superclasses. An object is an individual instance of a class. Message passing is a means of communication among the objects. When a message is received by an object, its appropriate method is invoked. Encapsulation is the process of isolating all the aspects of a class within its outline to ensure the protection of internal methods of the class and to give a clear interface with other classes. Polymorphism is the ability to have different methods in different classes that can handle the same messages.

An application of "frame" to the representation of the design standards organization has been proposed by Garrett [Garrett 86]. His classification system is built from three subject areas, or fields: object, stress-state, and limit-state. The "object" field is subdivided into two hierarchical trees, or facets: object-type and object-composition. Thus, there are four trees (two for the "object" field, one for stress-state, and one for limit-state). Each tree is composed of classifier frames. Each classifier frame has the following slots, such as "name" which represents the classifier and "<relation> <parent-frame>" which represents the relation between the classifier itself and its <parent-frame>. To build the classification system, the four trees shown in Figure 2-6 are merged to form a tree of classifier frames. Standards requirements are inserted into the classifier tree at the appropriate leaf classifier frames. A part of the classification system is shown in Figure 2-7.

To identify applicable requirements for a design member, the user gives partial features, or facets about the member, e.g., (OBJECT-TYPE, steel, I-shaped, hot-rolled, column, STRESS-STATE, axial, compression). The program then determines all the classifiers by

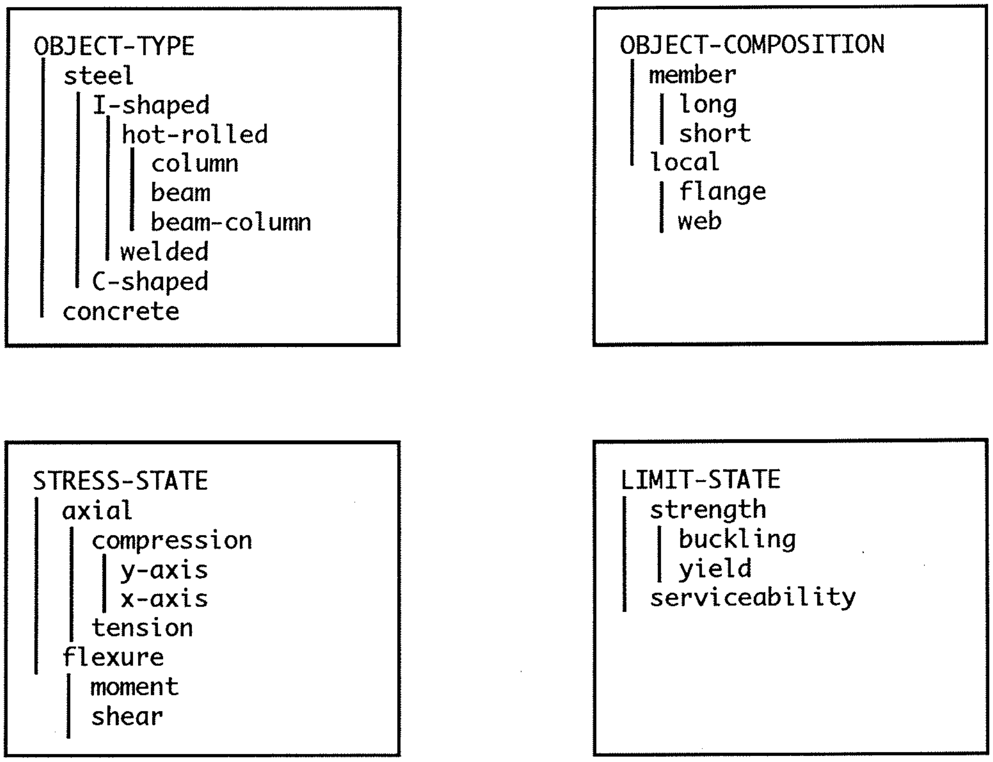


Figure 2-6 Partial Hierarchical Trees of Classifiers for Each Field [Garrett 86]

```
treetop
-- OBJECT-TYPES
---- steel
------ I-shaped
-------- hot-rolled
---------- column
------------- STRESS-STATES
-------------- axial
---------------- compression
------------------ y-axis
--------------------- LIMIT-STATES
---------------------- strength
------------------------ buckling
---------------------------- OBJECT-COMPOSITION
---------------------------- member
------------------------------ long
-------------------------------- REQUIREMENTS
                                  lrfd-column-buckling-y-axis-long
------------------------------ short
-------------------------------- REQUIREMENTS
                                  lrfd-column-buckling-y-axis-average
------------------ x-axis
--------------------- LIMIT-STATES
---------------------- strength
------------------------ buckling
---------------------------- OBJECT-COMPOSITION
---------------------------- member
------------------------------ long
-------------------------------- REQUIREMENTS
                                  lrfd-column-buckling-x-axis-long
------------------------------ short
-------------------------------- REQUIREMENTS
                                  lrfd-column-buckling-x-axis-average
---------------- tension
---------- beam
---------- beam-column
-------- welded
------ C-shaped
---- concrete
```

Figure 2-7 A Part of the Classification System [Garrett 86]

traversing the subtrees from the the given facets and executes all the requirements at the leaf classifiers. Within a decision table representing a requirement, the condition part contains a set of applicability criteria, which determine whether the requirement is applicable to a given member. If it is not applicable, its performance criteria are not executed and the value of that requirement is determined as "not applicable."

One deficiency in most methods representing design standards is treating each data item as variable for storing its value. Information about how to compute the value of a data item is kept separated from the data item. An object-oriented approach represents all the data items as separate, unique objects, each of which contains a method for determining its own value [Garrett 89].

In this section, a frame-based approach for representing the standards organization and an object-oriented approach for representing data items and methods were reviewed. In this thesis, the object-oriented paradigm is adopted for representing the organization of design provisions and design members.

2.1.4 Logic-Based Approach

2.1.4.1 Predicate Calculus

Predicate calculus is a formal language that provides a way of representing knowledge of objects and their relationships in the application domain of interest. The syntax of predicate calculus consists of two types of symbols: variables and constants. In the notation used most commonly, a variable starts with a lower case letter (e.g., x, y, z) and a constant starts with an upper case letter or a number (e.g., A, B, 36). Facts are stated in the form of expression called sentences, or well-formed formulas (wffs). There are three types of sentences:

- atomic sentences, which are formed from a *n*-ary relation constant with *n* terms, e.g., Supported_by (Beam_12, Column_34),
- logical sentences, in which atomic sentences are combined with logical operators, e.g., Section(Column, Compact) → Limit_state(Member_buckling), and
- quantified sentences, which have either universal or existential quantifiers, e.g., ∀x(∃y Supports(x,y)).

Semantics of predicate calculus can be defined as an evaluation whether the sentence accurately describes the world according to the conceptualization. An interpretation I is a mapping between elements of the language and elements of conceptualization. A variable assignment U is a relation from the variables of a language within the objects in the universe of discourse. If a sentence is satisfied by an interpretation I and a variable assignment U, the sentence is true with respect to the interpretation I and the relation U.

Inference is the process of deriving conclusions from premises. Given a set of predicate logical sentences, we can derive a conclusion by using a powerful inference rule known as the resolution principle. The resolution procedure requires the predicate logical sentences to be converted to a simplified form, called clausal form, e.g., {Section(Compact) ∨ Section(Noncompact) ∨ Section(Slender)}.

To derive a logical conclusion, the resolution procedure combines unification and elimination in a single operation. Unification is the process of determining whether two expressions can be made identical by the substitution of their variables. The basic form of the elimination rule is

$$((A \vee B) \wedge (\neg A \vee C)) \rightarrow (B \vee C).$$

where A is a literal, which is an atomic sentence or the negation of an atomic sentence, and B and C are clauses.

If a set of clauses is inconsistent, then it is always possible by resolution to deduce an empty clause. Thus, we can prove the inconsistency of a set of clauses by concluding an empty clause by resolution. We can establish that a set of formulas Δ logically implies the formula Ψ by showing that the combined formulas of Δ and the negation of Ψ, $\Delta \cup \{\neg\Psi\}$, is inconsistent. This method is called resolution refutation [Genesereth 87].

2.1.4.2 Logic Programming and Prolog

The use of predicate calculus as a programming language is called logic programming [Amble 87]. Logic programming is an attempt to store the knowledge of interest as a set of Horn clauses and to automate the process of deducing the answers by the resolution principle. A Horn clause is a variant of the predicate calculus. A Horn clause has a form of:

$$q \text{ :- } p_1, \ldots\ldots\ldots\ldots\ldots, p_m. \qquad m \geq 0$$

A sentence of this form says that q must be true if all of the p_i's are true. Given a set of clauses, we can logically derive a conclusion by the resolution principle.

Prolog is by far the most popular and known logic programming language. In Prolog, a constant starts with a lower-case character or a number while a variable starts with an upper-case character. The Prolog program must be a set of Horn clauses, which can be converted from clauses of predicate calculus. When issuing a query, which is a Horn clause, to the Prolog program, the Prolog interpreter attempts to deduce that query clause from the program.

Prolog is often viewed as a procedural programming language by ignoring its logical aspects. However, by separating the logical aspects from non-logical ones in the problem, one can use the logic programming paradigm in the development of a Prolog program [Deville 90]. In the implementation of HyperLRFD++, the logical aspect of Prolog language has been retained as much as possible by eliminating non-logical features such as cut "!" and the lack of "occur check" [Amble 87].

2.1.4.3 Logic-Based Approach for Standards Processing

A constraint-based approach for component design and conformance checking has been proposed by Chan [Chan 86]. Design specifications and device causality can be represented as constraints and implemented in Prolog. In the work, constraints are used:

- to derive a design description by propagating constraints with supplementary heuristics, and
- to check the design description if all the design parameters are given.

In a recent application, Rasdorf et al. [Rasdorf 90-b] [Lakmazaheri 90] show how logic can be used for both conformance checking and sizing of structural members. Standards provisions and a design situation are first represented as logical axioms, then a theorem prover performs design checking and member sizing. This formal model can be implemented by using a constraint logic programming language.

For the formulation and analysis of information by the writers of building regulations, a Prolog-based system has been proposed by Stone et al. [Stone 87]. In the framework proposed, the rule-base, which is a set of Prolog rules translated from the design provisions, is analyzed in terms of the rule dependency network and completeness and uniqueness. One of the limitations of this system is that uniqueness cannot be completely checked. Given a design description, the system first generates a conclusion. Then, the system backtracks to seek other solutions. If another solution is found, the rules lack in uniqueness. Since this method depends on sample design descriptions, it cannot prove the uniqueness, although lack of uniqueness may be found. The rule-base is used for compliance checking with a CADD system interface. During the conformance checking the user can ask "why" questions to the system. The answers to such questions are paraphrased rules in English, which are poor in providing justifications for the program and the design regulation. In the implementation of the Hyper-Object-Logic model, the design program is integrated with a generic document storage and retrieval system to facilitate finding such justifications for the user.

An application of predicate calculus to checking completeness and uniqueness of a design specification has been proposed by Jain et al. [Jain 89]. In this work, limited-entry decision tables are converted into predicate calculus sentences. Specifications are represented as groups of statements of the form S_i: $L_i \rightarrow R_i$, where L_i represents the part to the left of the implication while R_i represents the part to the right of the implication. Each group represents rules for a single data item. For example, the decision table for the determination of F_{cr} in Figure 2-2 can be converted as follows:

S_1: $C_1 \rightarrow A_1$

S_2: $\neg C_1 \rightarrow A_2$

Formal tests for checking completeness and uniqueness (lack of redundancy and lack of contradiction) of a group of rules representing a provision of the design standard based on predicate calculus are represented in this framework. This methodology is adopted and extended in the implementation of Hyper-Object-Logic model.

Formal logic has been used for processing (reasoning about) the SASE organizational model of a standard [Rasdorf 90-a]. The organizational submodel of a standard is represented as a set of classifier subtrees, a set of provisions, and the mappings between the subtrees and the provisions, as shown in Figure 2-8. Predicate logic is used to model this organizational submodel. The four properties of the organizational submodel include:

- abundant: a provision is abundant if it is mapped to multiple nodes within a subtree;
- free: a subclassifier, which is a complete path from the root node to a leaf node of a subtree, is free if it is not connected to any provision;

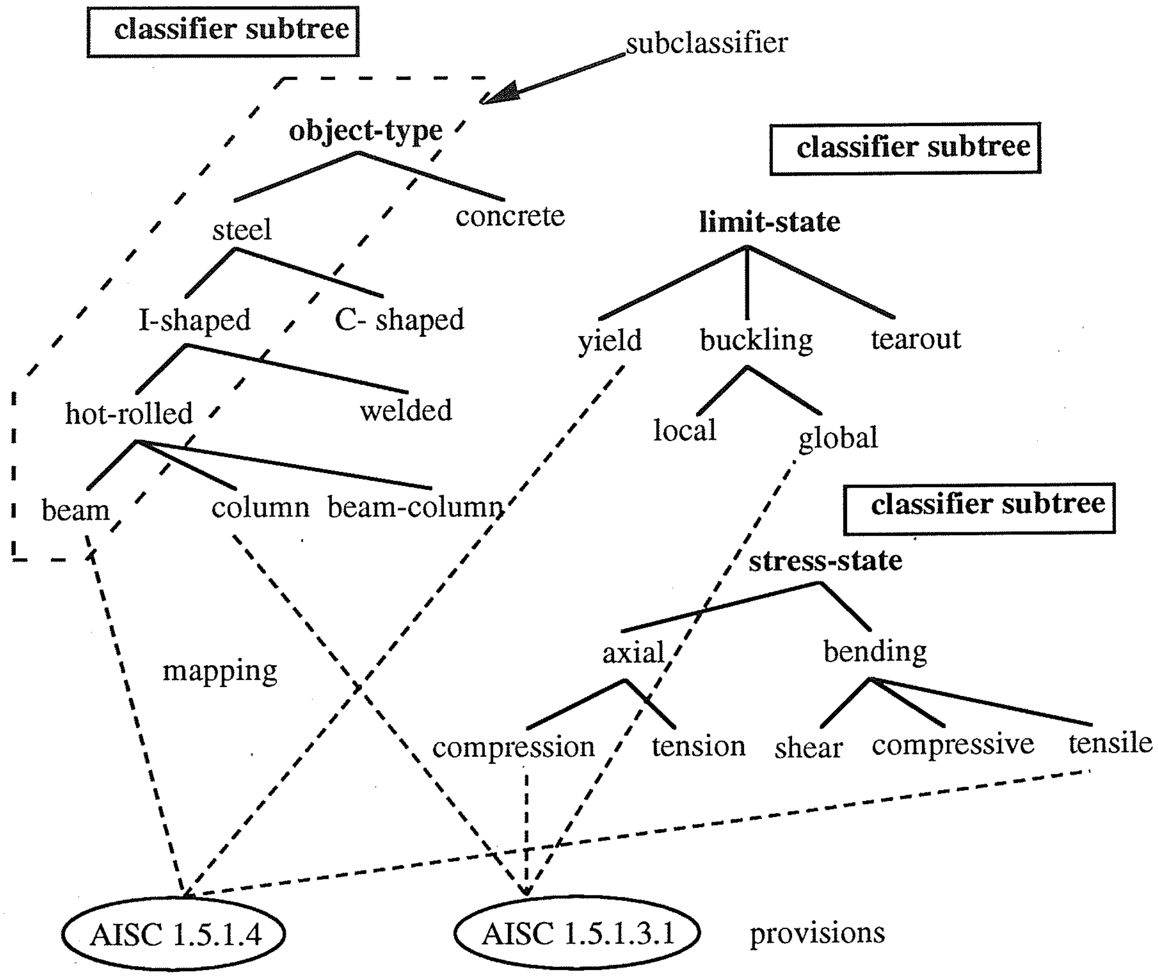


Figure 2-8 Organizational Submodel [Rasdorf 90-a]

- complete: a provision is complete if the number of mappings is equal to the number of subtrees; and
- unique: a provision is unique if it is complete and not abundant.

These four properties can be checked by the resolution theorem proving strategy in the work. On the other hand, the object-oriented paradigm seems more natural for representing the standard organization. Furthermore, by merging the classifier subtrees to make a unified tree, the four properties as noted above can be ensured easily.

2.1.5 Automated Component Design Using Standards

Conformance checking is a passive use of design standards for compliance. Several researchers have investigated how to generate component design by active use of design standards. Conformance checking is generally easier than producing a design description, because conformance checking has only two possible solutions (i.e., satisfied or violated), while multiple solutions could exist in a design problem.

One approach to automating component design uses symbolic algebra [Holtz 82]. In this method, decision tables are used to represent design provisions. The symbolic reformulation system converts constraints, derived from standards requirements as decision tables, into allowable boundaries of certain basic data items, called "designable" data. If only one data item does not have a value, the system produces numeric bounds on the value of that data item. Any value within the bounds satisfies the requirements of the standards. The user can choose a value from the feasible region. However, if more than one data item do not have values, the user has to select the designable data and determine how to produce boundary values for the data items that appear on the symbolic expression.

Another approach is to use a database management system (DBMS) for assigning attribute values such that the applicable design constraints are thereby satisfied if the constraints are expressed as equality [Fenves 85]. For inequality constraints and multiple constraints, some other procedures such as constraint re-formulation are necessary.

A numerical optimization technique has been used to generate component design using standards represented by decision tables [Garrett 86]. It was reported that the optimization routine may fail to find a solution when one actually exists.

One logic-based approach for designing is to let the theorem prover produce a predefined ID number of a component of design interest from a set of constraints of the design standard and a set of logic sentences depicting a design situation [Rasdorf 90-b] [Lakmazaheri 90]. ID numbers can be shape designations such as "W14x43" of W shapes of AISC manuals [AISC 89] [AISC 86]. The advantage of this approach is that the same logic description of the specification can be used for both conformance checking and generating component design. Although this approach itself is formal, if there exists a very large number of ID numbers of a component, this method could be inefficient.

Another methodology for component design using logic is to use heuristics for generating the trial values which are then tested against the relevant constraints [Chan 86]. The design heuristics are represented as constraints and written in Prolog. If the trial value does not satisfy the constraint, Prolog backtracks and return an alternative value. When all the possible values are exhausted, the heuristic simply fails.

In the Hyper-Object-Logic model, the heuristics-based approach is adopted. The primary reason for the heuristic-based approach is the efficiency because the number of standard shapes tends to be very large. The secondary reason is its applicability to arbitrary shapes such as non-standard, or customized components. In the theorem prover based approach, all the shapes must be predefined, while the heuristics could contain a procedure that generates a plausible arbitrary shape based on the given design situation.

2.2 Hypertext and HyperFile

Large heterogeneous document storage systems and document processing models are examined in this section. Section 2.2.1 discusses hypertext. In Section 2.2.2, the standard generalized markup language (SGML) and Hypermedia/Time-based document structuring language (HyTime) are examined. Section 2.2.3 introduces HyperFile and assesses the feasibility of its application to design standards documentation and processing.

2.2.1 Hypertext

Hypertext can be defined simply as the creation and representation of interlinked discrete pieces of data (text) [Fischer 90]. If this data is either an image or sound, in addition to text or numbers, the resulting structure is referred to as hypermedia [Parsaye 89]. Hypertext is composed of nodes, links, and a navigation system. Nodes represent discrete pieces of information and each of them can be shown on a screen or a window. The user can traverse between nodes via links, which are pointers from a node to other nodes. Links can be divided into two types: navigation links and organizational links. Navigation links connect a document to other referencing documents, including additional comments or explanations to the document. Organizational links connect the table of contents and indices of a book such as a design standard to its chapters, sections, or subsections, and connect a document to the following one, e.g., a link from Section 1.1 to Section 1.2. A navigation system provides a capability to find information in a systematic manner and with a minimum amount of effort. The tools to navigate through hypertext (hypermedia) are divided into three categories:

- links that allow traversal among the linked nodes,
- query languages that allow the user to request a list of documents filtered by queries based on keywords, and
- a browser that allows a map of all or portions of nodes and links, path history of previously visited nodes [Cornick 91].

The hypertext structure of directed linkage of documents is similar to semantic networks, where the nodes correspond to concepts and the links correspond to semantic relationships [Conklin 87]. The difference is that semantic networks are used to represent knowledge for inferencing, whereas hypertext is used to capture documents without regard to their machine interpretability. Hypertext can also be extended with frame-based or object-oriented systems.

The use of hypertext to design standards documentation has been proposed simultaneously by Cornick and Malasri et al. [Cornick 91] [Malasri 91]. Each section or provision of the standard is represented by a node. A table of contents is also a node, in which the user can go to any provision by clicking the provision title. If a provision contains references such as other provisions, tables, and charts, the user can go to the reference documents by clicking the name of the reference. In the work by Malasri et al. [Malasri 91], the earthquake section of the Uniform Building Code [ICBO 88] is implemented by using the hypertext software called GUIDE [Owl 90].

There are various advantages in using the hypertext approach for the documentation of design standards:

- The user can access desired information without viewing unnecessary information.
- The code can be easily updated and previous code can be attached.
- Explanations can be attached to provisions.

- Hypertext can be expanded to include video sequences for providing explanations.
- It can be interfaced with external programs, such as design and analysis programs.

On the other hand, there are also disadvantages to this approach:

- The hypertext version of design standards is limited by the platforms (software and hardware) selected by the developer.
- Users tend to lose their sense of location and direction in a complicated network of document nodes.

The approach described above uses hypertext statically, i.e., the nodes and links within the hypertext-based system cannot be changed by the user. Mitusch proposed "the spreadsheet approach" by applying dynamic use of hypertext for Norwegian building regulations [Mitusch 91]. All input and output data are represented as slots in multiple cards of HyperCard [Apple 90]. Similar to spreadsheet programs, when the user specifies the input data such as floor heights, floor areas, activities, and escape routes, the embedded programs written in HyperTalk computes desired results and show it on the card. The input data can be changed at any time and the consequences according to the regulations are updated. This dynamic hypertext approach enables the user to perform design and design checking. However, since the embedded programs are "hard-coded," it is difficult to modify and check correctness of the program.

2.2.2 The Standard Markup Language and HyTime

The Standard Generalized Markup Language (SGML) is the International Organization for Standardization (ISO) standard for document description (ISO 8879) [Goldfarb 90]. It is designed to enable text interchange among different document processing software and hardware. The standardized markup specifies document structure and appearance, e.g., the organization of the document, formats, and fonts. The markup is a tag for the information about the structure and notation(s) of the document. By reading the markup any application software that has been provided with an appropriate data converter can understand and interpret the document. An application of the SGML to design standards has been proposed by Bourdeau [Bourdeau 91]. Texts and tables of the French building technical rules are keyboarded following the SGML and drawings are vectorized or just scanned in the project. These documents are stored in a CD-ROM (Compact Disk - Read-Only Memory) and can be read by using hypertext software.

Another standard Hypermedia/Time-based Document Structuring Language called "HyTime", using the SGML, has also been proposed [Newcomb 91]. HyTime is a draft standard language for representing the structure of multimedia, hypertext, hypermedia, time- and space-based documents (such as music documents and icons that their relative positions are specified on the document). It allows hypermedia software which cognize HyTime to browse, render, format, and query the documents compliant to HyTime even if that software is not able to understand or render its multimedia objects.

The SGML and HyTime have many advantages and seem to be a promising means for document publishing, storage, and conversion. However, SGML use has significant costs [Van Herwijnen 90], and very few word processors are SGML compatible. For the reason of practicality, only the concept of standardized markups and pointers to other documents of the SGML and HyTime are considered in the Hyper-Object-Logic model.

2.2.3 HyperFile

HyperFile is a back-end data storage and retrieval facility for heterogeneous document management applications [Clifton 90]. The goal of HyperFile is:

- to store not only traditional documents containing text but also multimedia documents containing images, graphics, or audio,
- to support hypertext applications, and
- to provide a shared repository for multimedia and diverse applications.

HyperFile is intended as a back-end service as shown in Figure 2-9. The HyperFile structure allows the user running a particular document management system to view a design drawing stored in HyperFile. Similarly, a user running a design tool should be able to refer to a document that describes the operation of a particular component.

In HyperFile, objects represented as files are modeled as sets of tuples. These tuples can contain text, pictures, key words, references and pointers to other objects, or arbitrary bit strings. This simplicity of the structure makes HyperFile very useful for diverse applications. Tuples have three parts:

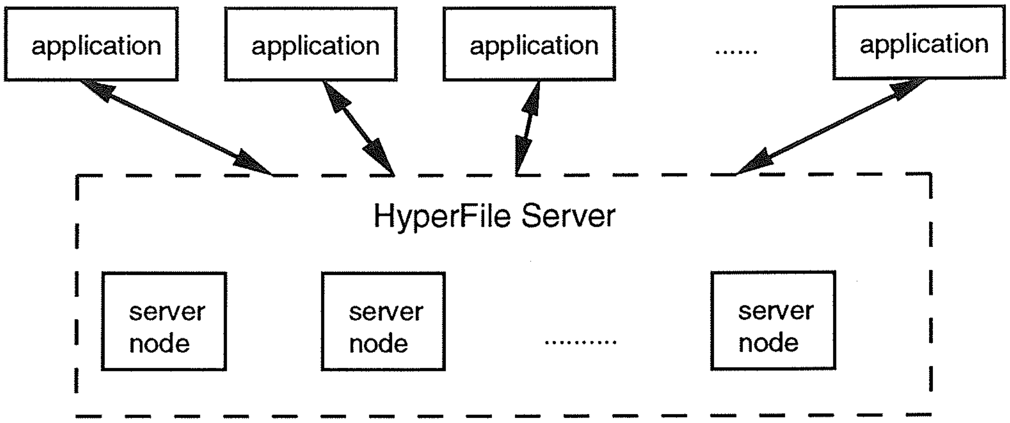


Figure 2-9 HyperFile as a Back-end Service [Clifton 90]

- type, which identifies the data types of the remaining fields to HyperFile,
- key, which is used by the application to specify the purpose of the tuple, and
- data, which can be simple type such as a string or pointer, or complex (and not understood by HyperFile) such as a paragraph of text or the object code of a program.

A sample set "E2," containing (for example) Section E2 of the AISC LRFD specification [AISC 86], is:

```
{ (String,     "Title",          "E2. DESIGN COMPRESSION STRENGTH")
  (String,     "Author",         "AISC")
  (String,     "Keyword",        "column")
  (String,     "Keyword",        "compression")
  (String,     "Keyword",        "buckling")
  (Text,       "Description",    <The design strength of compression
                    member whose elements ...., see Appendix B5.>)
  (Pointer,    "Sect. B5.1",     <Pointer to Sect. B5.1.)
  (Pointer,    "Appendix B5.3",  <Pointer to Appendix B5.3>)   }
```

HyperFile queries are based on the browsing techniques of hypertext with the addition of a query language based on document sets and filtering. These queries consist of three basic parts:

- A starting set of objects in the graph-structured document repository,
- A set of filtering criteria (keywords, size, etc.),
- A description of where to look: what types of links to follow (and how far) to find prospective objects.

A sample query to find all objects in the set "E2" which have a keyword "effective length factor" is:

```
E2 | (String, "Keyword", "effective length factor") -> T
```

This query takes the objects pointed by E2, checks to see if they have a tuple of type String with the Key Keyword and data "effective length factor," and puts the resulting items into the set T. More complicated queries such as traversing the graph created by pointers can be given (see [Clifton 90]).

HyperFile overcomes the primary disadvantage of hypertext, being limited to the software selected. The HypeFile's powerful query facility also addresses the "lost in hyperspace" problem that arises in large hypermedia systems. In comparing with SGML and HyTime, the initial and development costs are very low due to its simple structure. HyperFile seems to be a promising framework for storing, retrieving, and managing a very large amount of heterogeneous documents including graphics, sound, and video as well as text. The concept of HyperFile has been employed in the development of the Hyper-Object-Logic model.

2.3 Summary

In Section 2.1, previous approaches for design standards representation and processing were described. The SASE model, which is based on decision tables, was first reviewed. Then, the recent new approaches, a production rule approach, frame-based and object-oriented representation, and logic based approach, were discussed. Each approach's representation technique and methodology of analyzing the design standard (if any) were reviewed. Finally, several approaches for automated component design using design standards were discussed.

The object-orientation appears to be a most natural and promising paradigm for representing the organization of the design standard for its hierarchical structure. This paradigm is also useful for representing each data item as a unique separate object with the method to determine its value. Design provisions can be represented either in decision tables or logic sentences. While it is difficult to automate component design in the decision-table-based approach, the logic programming paradigm can be used for both conformance checking and component design. It also lends itself to checking requisite properties of completeness and uniqueness of design standards. Thus, the logic programming paradigm is adopted for representing design provisions for this research. For implementation of the model, the combination of object-oriented and logic programming paradigms is used.

In Section 2.2, document storage and retrieval systems and models were described. Hypertext-based approach for standards representation was first reviewed. Then, the standard languages of SGML and HyTime were examined. Finally, HyperFile was discussed. HyperFile is an excellent structure for a large heterogeneous document storage and retrieval system. Furthermore, since HyperFile can make linkages between provision documents and program code representing provisions, it provides a means to check the requisite property of correctness of design standards. The form of the HyperFile structure has been employed in the development of the Hyper-Object-Logic model. Although the SGML and HyTime have many advantages, the development cost could be enormous. Only the concept of markups and pointers of the SGML and HyTime has been adopted in this work.

This thesis proposes a new framework by combining the standards processing system, based on the unified Object-Logic paradigms and the HyperFile structure. In the following chapters, the framework, the Hyper-Object-Logic model, and the prototype system called HyperLRFD++ are discussed.

Chapter 3
The Hyper-Object-Logic Model

This chapter provides an overview of a framework for the documentation, representation, and processing of design standards. There are two distinct categories of knowledge in a design standard:

- knowledge of the organization of design objects, and
- knowledge of the methods in reasoning about design.

The object-oriented paradigm lends itself naturally to representing the organizational aspect of the design standard. The logic programming paradigm, on the other hand, is well suited to implementing the reasoning mechanisms for design. The object-oriented and logic programming paradigms are combined to provide a unified Object-Logic model for the representation of design codes and the processing of the design standards. By storing the design provisions in a knowledge base, the model is capable of performing conformance checking and component design, and syntactically analyzing the standard.

Besides the Object-Logic representation, the framework also includes the storage of background information (such as commentaries) of the design provisions in a HyperDocument model, which is based on a form of HyperFile structure [Clifton 90]. The HyperDocument model allows code developers to store and access heterogeneous documents, including design provisions, their background information and data, and programs, for developing and revising design standards. The model also allows designers and engineers to obtain explanations and other background information to support design tasks.

The Object-Logic and the HyperDocument models are integrated into a unified framework. Section 3.1 presents an overview of the Hyper-Object-Logic model. Section 3.2 presents a summary of the capabilities of a prototype system developed based on the model. Section 3.3 describes an architecture of a prototype system of this model.

3.1 An Overview of the Model

A design standards processing system should have the following features:

- The user can perform both conformance checking and component design within the same design environment.
- The system should represent both the organization and provisions of the standard effectively so that applicable requirements can be automatically identified and executed.
- The properties of completeness, uniqueness, and correctness of the provisions can be checked in a systematic and automatic manner.
- Background information of the design standard should be stored and accessed easily if necessary.
- The system should help engineers to understand implications of the provisions.
- The system should be integrated with design applications and engineering databases.

The Hyper-Object-Logic model for the documentation, representation, and processing of design standards is developed to provide these features. The overall architecture of the Hyper-Object-Logic model is depicted as shown in Figure 3-1. Broadly speaking, this model consists of two submodels:

- Object-Logic model, and
- HyperDocument model.

The two models are integrated together by sharing Method Objects, which are program codes representing the design provisions, and are stored in the Standards Base. In the following subsections, an overview of the two models are provided. Detailed descriptions of the models are discussed in the succeeding chapters.

3.1.1 The Object-Logic Model

The Object-Logic model represents the organization and provisions of design standards, using a combination of object-oriented and logic programming paradigms. The model allows the user to perform both conformance checking and component design generation within the same design environment. The model also allows checking for anomalies such as incompleteness and inconsistency of the design standard.

The Object-Logic model consists of the following five modules:

- Standards Base, which represents the organization and provisions of the standard,
- CAD Object Data Model, which facilitates member definition using the Object Model and retrieving member data from engineering databases.
- Conformance Checking Module, which performs conformance checking of a given member,
- Component Design Module, which generates component design of a given member,
- Standards Analysis Module, which checks completeness and uniqueness of provisions and the organization of the standard, and also examines the relations of provisions,

The Standards Base has the following two sub components:

- Member Class Hierarchy, which represents the organization of the standard, and
- Method Objects, which represent the provisions of the standard.

The Member Class Hierarchy is an object-oriented organizational model of design standards, which is partially based on the frame-based classification system proposed by Garrett [Garrett 86]. The Member Class Hierarchy consists of classes, which contain attributes and pointers to Method Objects. Given a design member, the Member Class Hierarchy can be used to identify all the applicable provisions based on the attributes and properties. Each provision is represented by Object-Logic program in the Method Object. Reasoning such as concluding whether the given design member satisfies a requirement is done by logical resolution and message passing among Method Objects.

A design member object consists of attributes, or properties of the member, and external constraints given by the user. To define the attribute values and external constraints to the design member, a CAD Object Data Base can be used. The main components of the CAD

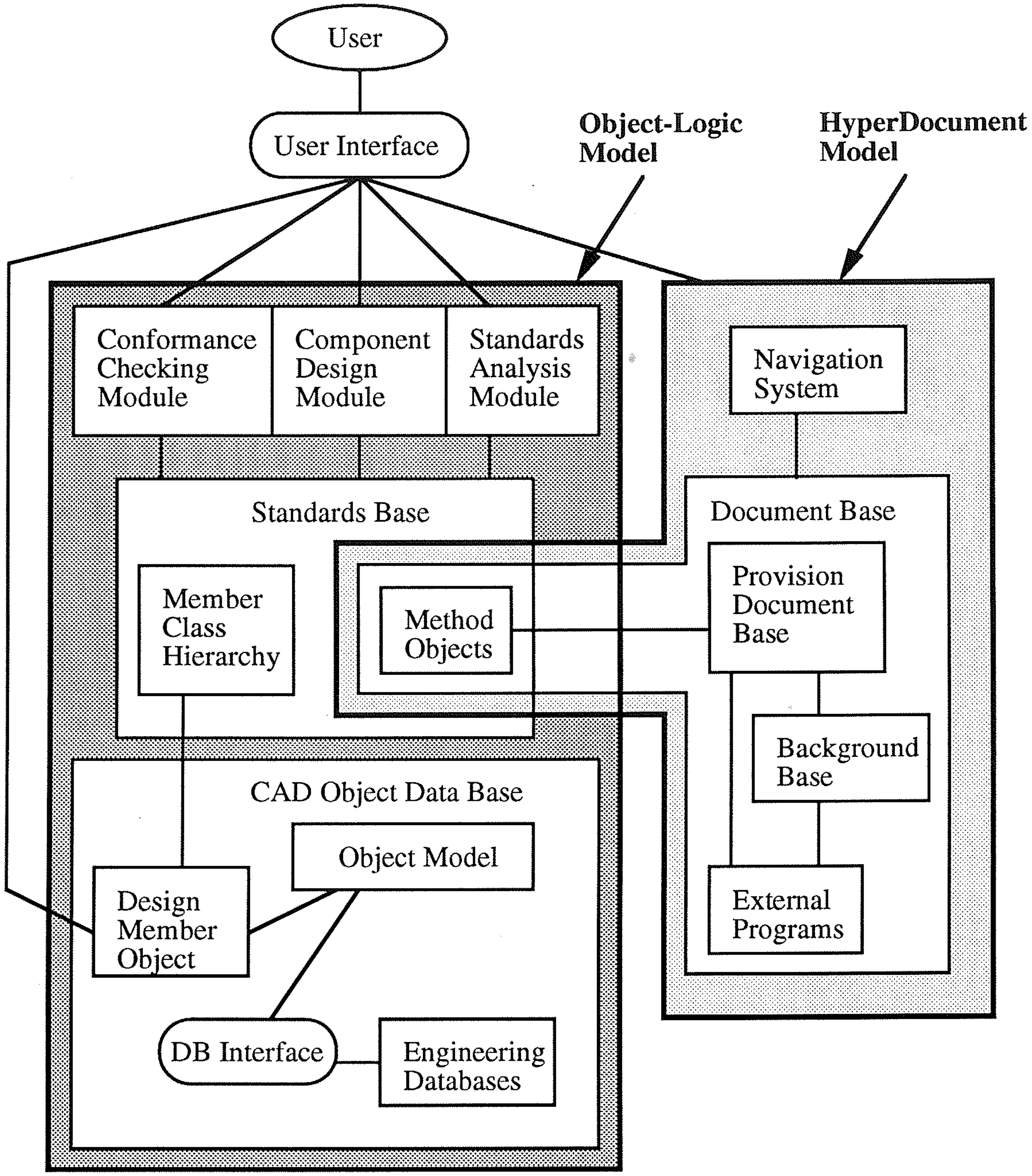


Figure 3-1 Overview of the Hyper-Object-Logic Model

Object Data Base are

- an Object Model, which is a hierarchical structure representing member objects, and
- engineering databases, which contain data that are not often included in the standard such as dimensions of standard component shapes.

If an attribute value of the design member object is not given, the data are retrieved from the engineering database, if available.

The Conformance Checking Module checks a given design member for conformance with applicable requirements. The Component Design Module generates a trial design description based on a set of design heuristics and checks whether the candidate satisfies the applicable requirements. If the trial member does not satisfy the design requirements, the module generates another candidate and checks for compliance.

The Standards Analysis Module checks completeness and uniqueness (the lack of redundancy and the lack of contradiction) of a set of rules in the Method Object. The testing method is based on the procedure proposed by Jain et al. [Jain89]. Code developers can check the three required properties at both the provision and the organization levels. It also checks whether the relationships among the Method Objects are connected and acyclic.

3.1.2 The HyperDocument Model

The HyperDocument model contains the provisions, background information of standards, external programs, and Method Objects. The documents and programs stored in the HyperDocument model can help engineers to design through easy access to external programs and embedded Method Objects and to serve as a large document storage system for the design provisions and background information.

The HyperDocument model consists of a Document Base and a Navigation system. The Document Base contains the documents, each of which consists of its content and HyperTag. The HyperTag is a tag that contains information about the document such as title, author, file name, rendering software, and pointers to other documents. The content of each document is shown and processed by its rendering software such as word processing software. The Navigation system facilitates document retrieval, and provides three basic navigation mechanisms: pointers, queries, and browsers.

The Document Base of the HyperDocument system for design standards consists of the following four document clusters:

- Method Objects, which are Object-Logic programs representing provisions,
- Provision Document Base, which contains provision documents,
- Background Base, which contains a large amount of background information related to design standards such as explanations and commentaries to the provisions and research papers, and
- External Programs, which process charts or complex equations that appear in the standard and background information to derive necessary data items.

When creating or revising a standard, a large amount of information in various forms may be collected. Such information as well as explanations about provisions is stored in the Background Base. The documents in the Background Base can be used by code developers for future revisions and by engineers for obtaining explanations and background information about the provisions.

During the design process, many data items require complex calculations involving graph or chart processing. External programs perform such tasks and return the computed results to the design program. Such programs, for example, spreadsheet programs, can be accessed through the HyperDocument system.

3.2 System Capabilities

This section briefly discusses the potential applications of the Hyper-Object-Logic model in structural design and development of design standards.

3.2.1 Structural Design

A typical structural design process can be depicted as shown in Figure 3-2. The design process can be divided into two phases: preliminary design and detailed design. In the

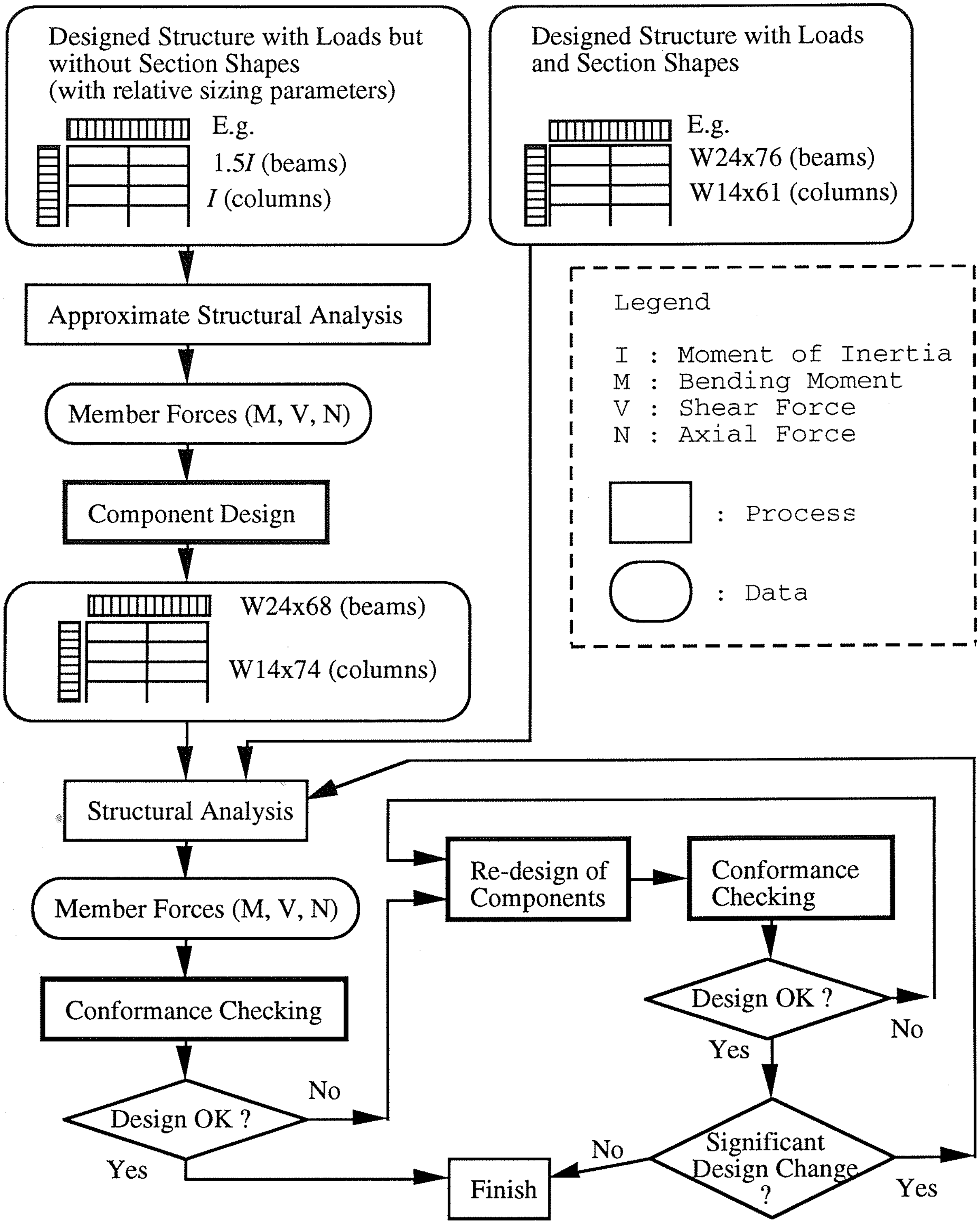


Figure 3-2 Structural Design Process

preliminary design phase, an initial configuration of the main structural components are defined and member forces are estimated using approximate structural analysis methods. Once the member forces are obtained, preliminary member properties can be selected. The member can then be checked to conform with design standards and constraints of the project. In preliminary design, not all applicable requirements of the design standard need to be checked. Engineers usually focus on one (or a few) requirement(s) that are likely to govern the design of the member. If the member violates the requirement(s), it is re-designed. If the modification in the re-design is significant, the structure should be re-analyzed.

In the detailed design phase, the configuration and properties of all structural components are often pre-determined. A detail structural analysis can be performed to determine the member forces. Based on the member forces, the components need to be checked against all the applicable requirements. If a member violates certain design requirements, an iterative design and analysis process is needed until all requirements are satisfied.

Although their basic purposes and levels of performance are different, both preliminary and detailed design phases have certain similar features, which can be represented in the flowchart as shown in Figure 3-2. The objectives of the Hyper-Object-Logic model in the design process are: to facilitate the preliminary design of components, conformance checking with respect to applicable code requirements, and re-design process. The Hyper-Object-Logic model can be used to enhance the design process as shown in Figure 3-3, where the design tasks supported by the model are depicted. The Conformance Checking Module supports conformance checking, the Component Design Module supports the component design and re-design of components, and the HyperDocument model supports the re-design of components by users. Illustrative examples of design applications of the Hyper-Object-Logic model are given in Chapter 6.

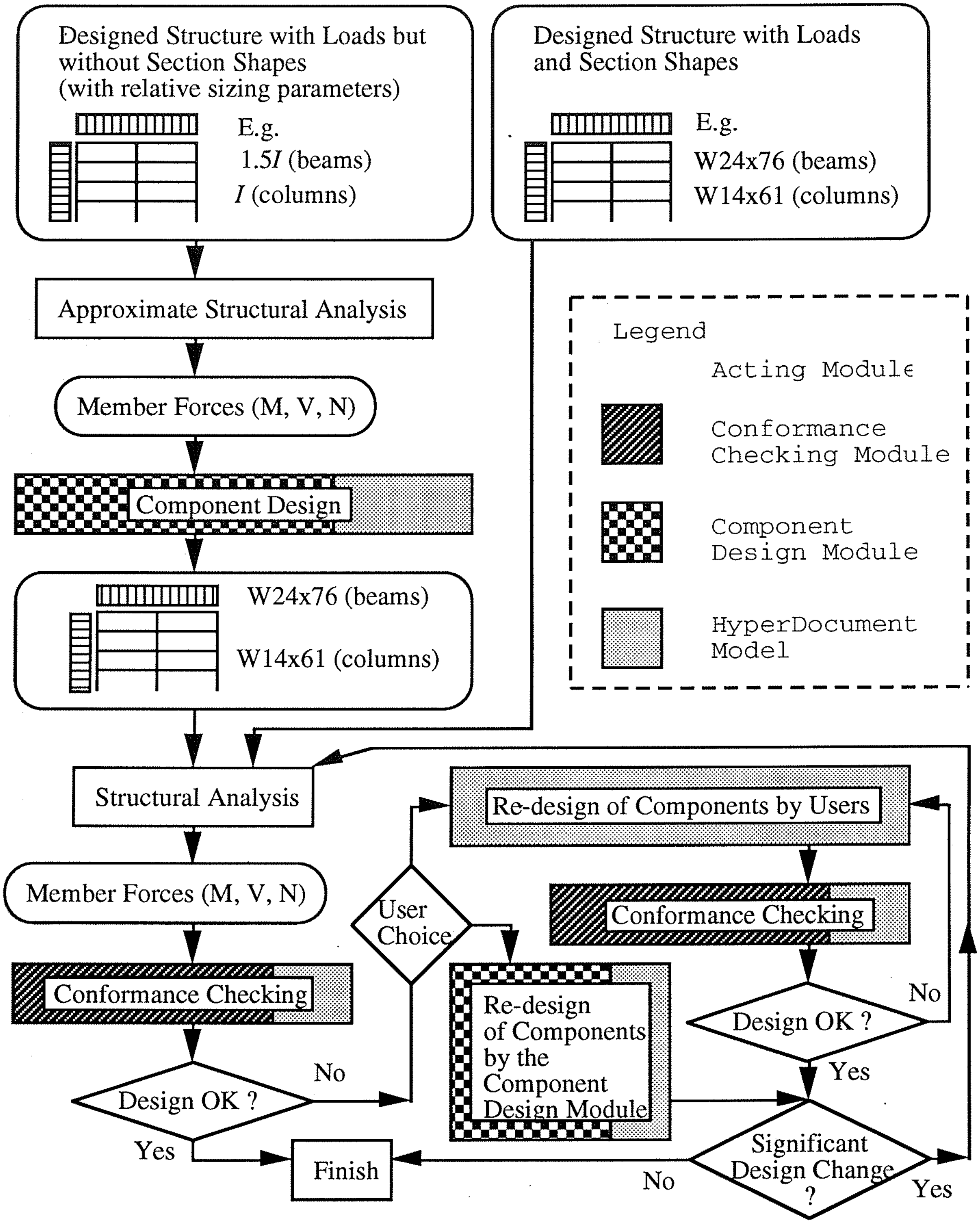


Figure 3-3 Structural Design Process and the Hyper-Object-Logic Model

3.2.2 Development of Design Standards

The design standard is usually developed by a technical committee consisting of up to tens of members. The development of a design standard is an iterative and tedious process. Figure 3-4 shows a simplification of the process. In general, the committee collects related documents and prepares a preliminary draft standard. The committee members discuss and review the draft documents until "consensus" can be reached. Throughout this process, many supporting data, background information, and documents are used but not recorded in the design specification. Furthermore, inconsistency among design provisions often occurs.

The Hyper-Object-Logic model can enhance the design standards development process, depicted as shown in Figure 3-5. The HyperDocument system provides code developers with the capabilities of collecting the related documents easily and quickly and sharing them among committee members. When the preliminary draft is rewritten, the committee stores the revised draft as Provision Documents in the HyperDocument system. When the draft standard is translated into Member Class Hierarchy and Method Objects, the Standards Analysis Module can be used to check the required properties of the standard at the provision level and the organization level. The mappings between the Provision Documents and Method Objects provide a capability of checking whether the Method Objects, i.e., the program code, are correctly interpreted from the provisions. Finally, the

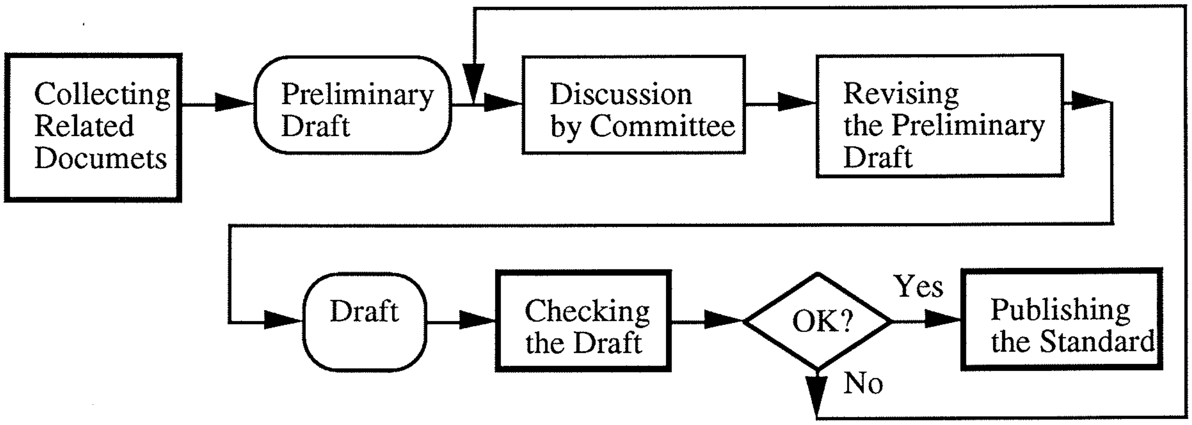


Figure 3-4 Process of Design Standards Development

provision documents in the HyperDocument model and the Member Class Hierarchy and the Method Objects in the Object-Logic model can be published with a book version of the standard so that they can directly be used for component design and conformance checking purposes.

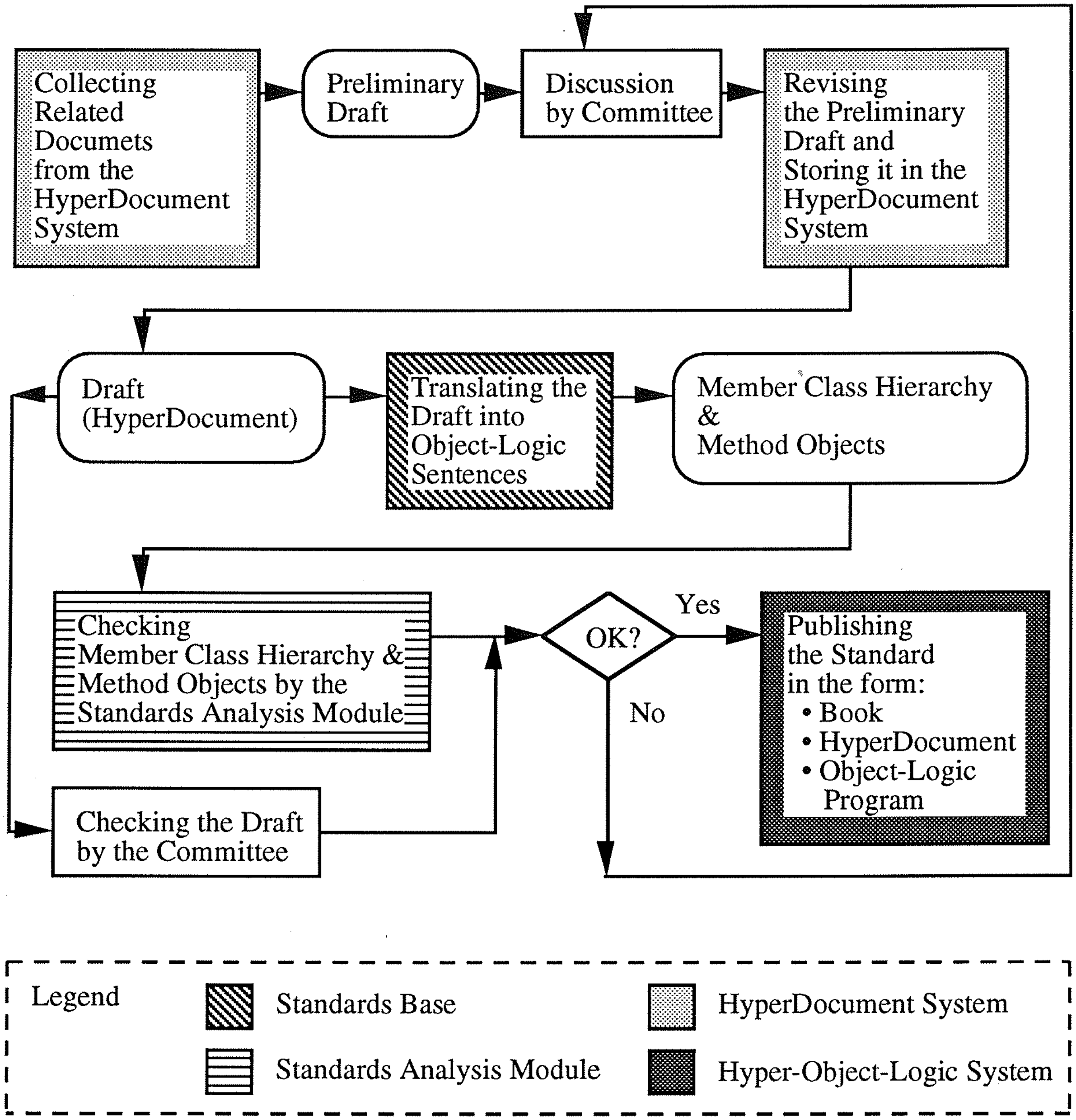

Collecting Related Documets from the HyperDocument System
Preliminary Draft
Discussion by Committee
Revising the Preliminary Draft and Storing it in the HyperDocument System
Draft (HyperDocument)
Translating the Draft into Object-Logic Sentences
Member Class Hierarchy & Method Objects
Checking Member Class Hierarchy & Method Objects by the Standards Analysis Module
OK?
Yes
No
Publishing the Standard in the form:
• Book
• HyperDocument
• Object-Logic Program
Checking the Draft by the Committee
Legend
Standards Base
HyperDocument System
Standards Analysis Module
Hyper-Object-Logic System

Figure 3-5 The Process of Design Standards Development and the Hyper-Object-Logic Model

3.3 The Prototype System

To demonstrate the feasibility and practicality of the Hyper-Object-Logic model, a prototype system, HyperLRFD++, has been implemented for the AISC LRFD specification [AISC 86]. HyperLRFD++ incorporates the sections of the AISC LRFD specification that are related to the following member types:

- compression members,
- flexural members, and
- members subject to bending and compression (i.e., beam-columns).

For demonstration purpose, only wide-flange W sections are included in the database. HyperLRFD++ is capable of component design and conformance checking of W shape section members. Furthermore, the prototype system can be used to analyze the design standard.

HyperLRFD++ has been implemented on an Apple Macintosh IIsi computer. The following software packages are employed in the implementation:

- Prolog++ [Quintus 90], an extension of Prolog with a full object-oriented programming environment, is used to implement the Conformance Checking Module, Component Design Module, Standards Analysis Module, Standards Base, Object Data Model, and Member Object.
- HyperCard [Apple 90], a Hypertext software for Macintosh, is used to implement HyperTags and documents in the HyperDocument system and the user interface.
- Oracle [Oracle 89], a relational database system, is used to implement the dimensions and properties of wide-flange W shape sections.
- MacDBI [Quintus 91], a system interface between Prolog and the Oracle database system for Macintosh, is used to integrate the Object Data Model and the engineering databases within the CAD Object Data Base.
- Excel [Microsoft 91], a spreadsheet software, is used to implement the external program that processes graphs and charts such as the alignment chart for determining the effective length factor.

Figure 3-6 shows the overall structure of the HyperLRFD++ and software packages that are used for implementation.

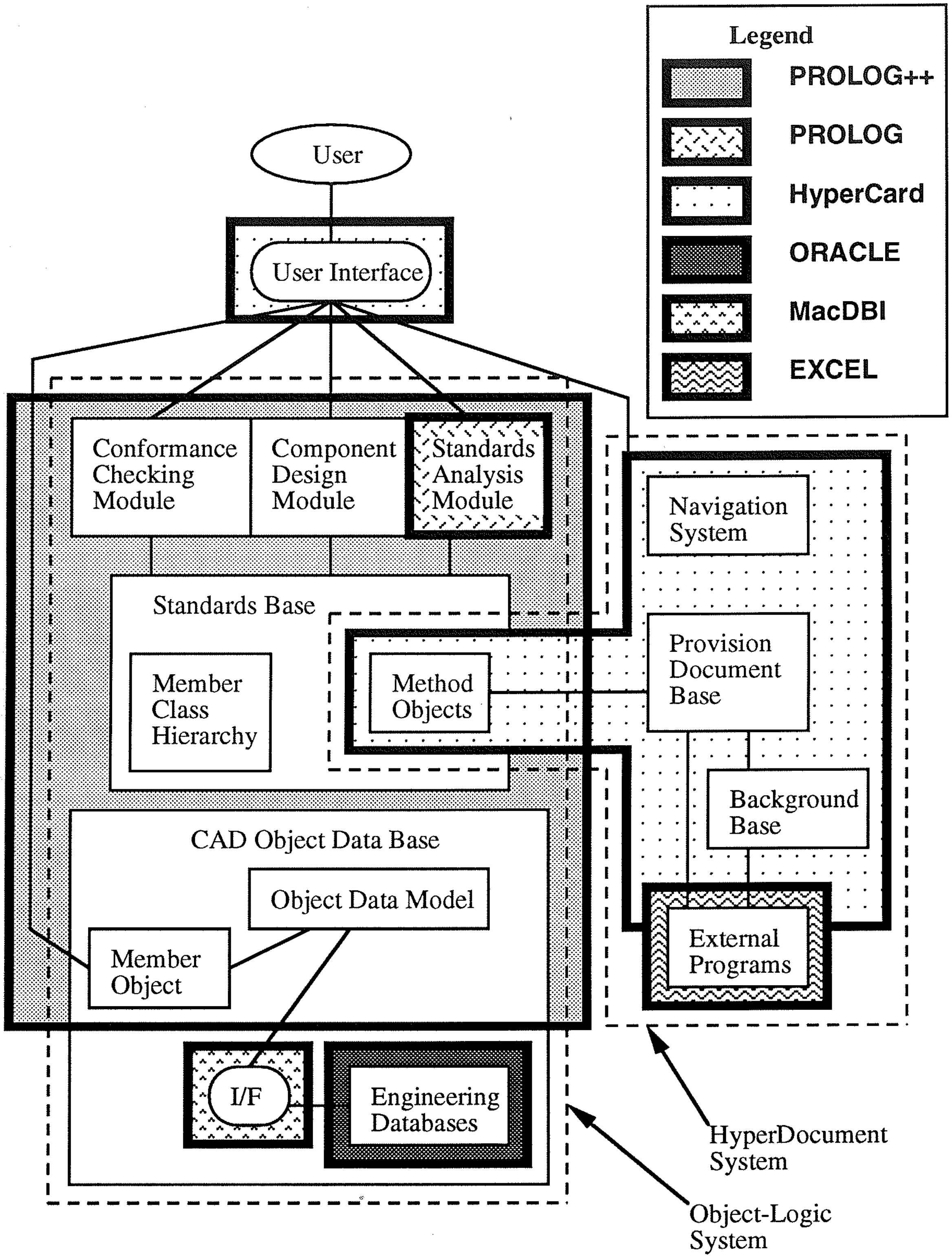


Figure 3-6 The Structure of HyperLRFD++ and Implementation Software

3.4 Summary

In this chapter, an overview of the Hyper-Object-Logic model was presented. The potential applications of the model for structural design and design standards development were described. An architecture of the prototype system, HyperLRFD++, based on the Hyper-Object-Logic model, was presented. Details of the model are described in the succeeding chapters. The next chapter describes the Standards Base and the CAD Object Data Base of the Object-Logic model. Chapter 5 presents the HyperDocument model. Chapter 6 provides the details on the Conformance Checking Module and Component Design Module and illustrates a few examples of design applications of the Hyper-Object-Logic model. Chapter 7 presents the Standards Analysis Module and illustrates a few examples of checking properties of design standards.

Chapter 4
The Object-Logic Model

This chapter describes two main components of the Object-Logic model: the Standards Base and the CAD Object Data Base, for the representation and processing of design standards. Figure 4-1 depicts the system architecture of the Object-Logic model. The system consists of five basic modules:

- Standards Base,
- CAD Object Data Base,
- Conformance Checking Module,
- Component Design Module, and
- Standards Analysis Module.

The details of the Standards Base and the CAD Object Data Base are described in the following sections. The Conformance Checking Module and Component Design Module are explained in Chapter 6. The Standards Analysis Module is discussed in Chapter 7.

4.1 Standards Base

The organization and provisions of the standard are represented and stored in the Standards Base. The organization of the standard is represented by Member Class Hierarchy and the provisions are implemented by Method Objects.

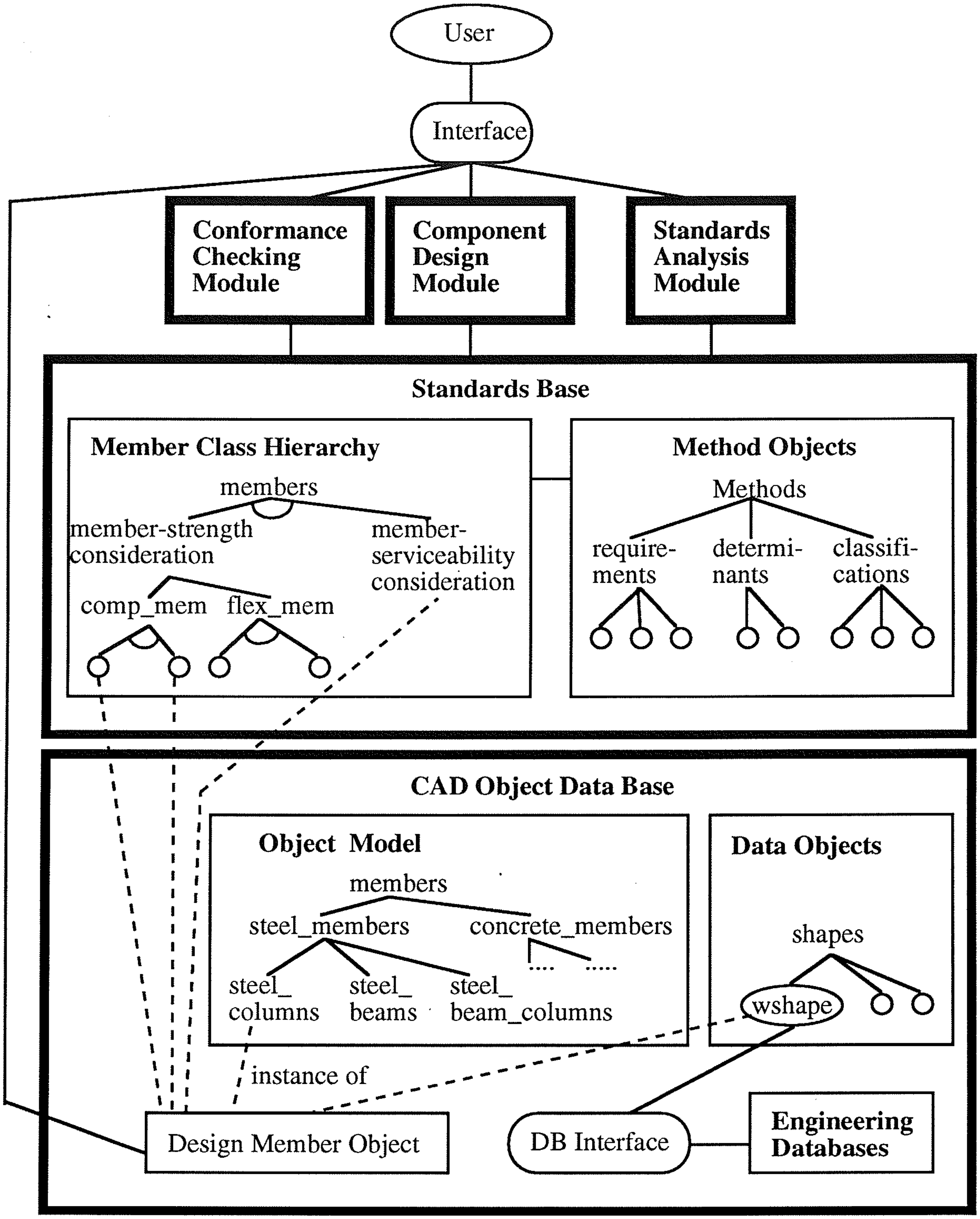


Figure 4-1 System Architecture of the Object-Logic Model

4.1.1 Member Class Hierarchy

A design standard includes a set of provisions that a design object must satisfy. The provisions provide the requirements that physical objects such as beams and columns must conform with as well as guidelines by which abstract concepts such as measurement and maintenance are prescribed. Typically, a standard has several chapters, each of which represents a class of objects. Each chapter may contain some general provisions about the class and is subdivided into several sections. Each section, which may contain many provisions, may be further subdivided into subsections. That is, a standard is often organized in a hierarchical manner. The root node of the hierarchy corresponds to the most general thing in the standard, e.g. members. Each node of the hierarchy represents a chapter, a section, or a subsection and contains specific provisions. Provisions at the higher level of the hierarchy are "inherited" by the nodes at the lower level. As an example, the outline of the content of the AISC LRFD specification for steel construction [AISC 86] is shown in Figure 4-2.

4.1.1.1 The Structure of Member Class Hierarchy

The organization of a standard can be represented in a hierarchical object model. One example of such a model is shown in Figure 4-3.

In the Object-Logic model, the organizational model of design standards is an object-oriented class hierarchy, partially based on the frame-based classification system proposed by Garrett [Garrett 86]. The model can be established using the following procedure:

1. Identify the fields and their classifiers of the standard. For example, Figure 4-4 shows three fields: object, stress-state, and limit-state, and their partial classifiers from the AISC LRFD specification [AISC 86].
2. Merge all the classifier trees obtained in (1) to develop a unified hierarchical tree, as shown in Figure 4-5. Delete inappropriate branches such as stress-state of compression within the tension member subtree.
3. Remove the names of the fields from the unified hierarchical tree and rename the classes. Figure 4-6 shows an object hierarchical tree that is converted from the tree in Figure 4-5. In this example, the nodes, "global buckling" and "local buckling," in the previous tree correspond to the classes, "nonslender_y_comp_mem" and

A. GENERAL PROVISIONS

....

B. DESIGN REQUIREMENTS

....

C. FRAMES AND OTHER STRUCTURES

....

D. TENSION MEMBERS

D1. Design Tensile Strength
D2. Built-up Members
D3. Eyebars and Pin-connected Members

E. COLUMNS AND OTHER COMPRESSION MEMBERS

E1. Effective Length and Slenderness Limitations
    1. Effective Length
    2. Plastic Analysis
E2. Design Compressive Strength
E3. Flexural-torsional Buckling
E4. Built-up Members
E5. Pin-connected Compression Members

F. BEAMS AND OTHER FLEXURAL MEMBERS

F1. Design for Flexure
    1. Unbraced Length for Plastic Analysis
    2. Flexural Design Strength
    3. Compact Section Members with $L_b \leq L_r$
    4. Compact Section Members with $L_b > L_r$
    5. Tees and Double-angle Beams
    6. Noncompact Plate Girders
    7. Nominal Flexural Strength of Other Sections
F2. Design for Shear
    1. Web Area Determination
    2. Design Shear Strength
F3. Transverse Stiffeners
F4. Web-tapered Members (see Appendix F4)

G. PLATE GIRDERS

H. MEMBERS UNDER TORSION AND COMBINED FORCES

....

I. COMPOSITE MEMBERS

....

J. CONNECTIONS, JOINTS AND FASTENERS

....

K. STRENGTH DESIGN CONSIDERATION

....

L. SERVICEABILITY DESIGN CONSIDERATIONS

....

M. FABRICATION, ERECTION AND QUALITY CONTROL

....

Figure 4-2 A Part of the Table of Contents of the AISC LRFD Specification [AISC86]

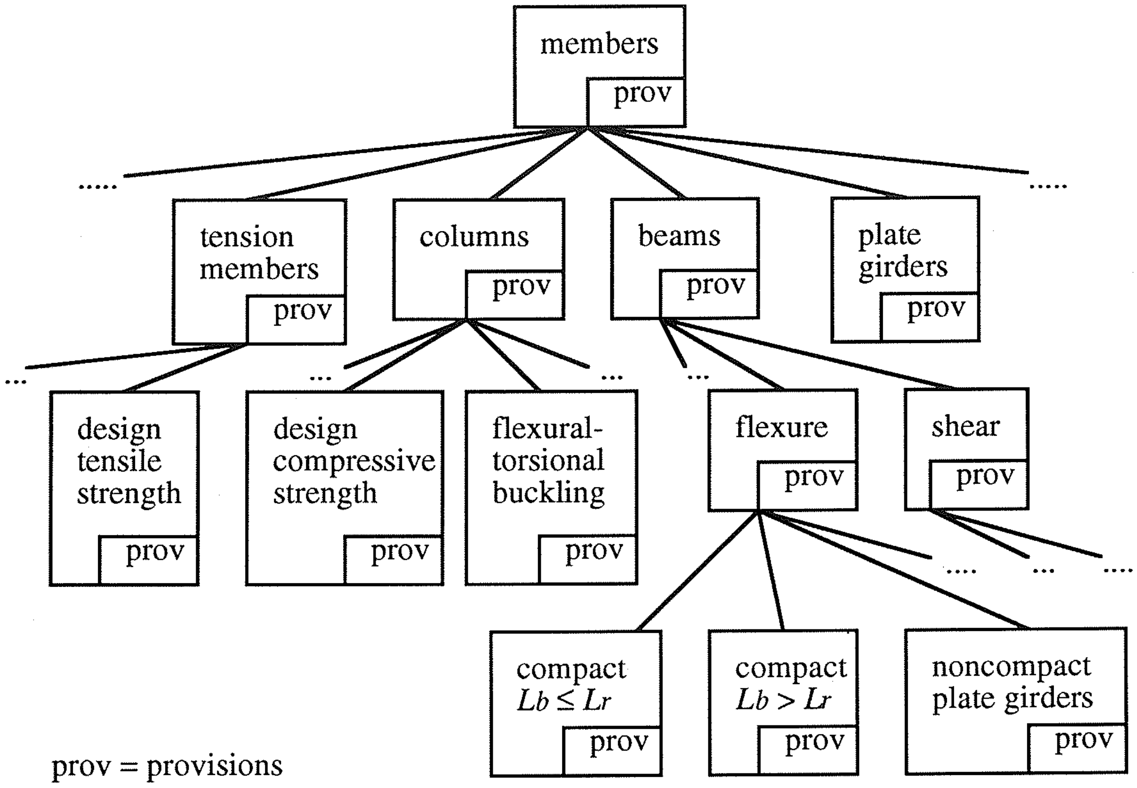


Figure 4-3 Object-Oriented Organization Model of the Standard

"slender_y_comp_mem," respectively.

4. Attach each requirement to only the corresponding leaf node class, and attach each determinant to a corresponding class so that its descendant classes can inherit the determinant.

Given a design member, the way to identify applicable provisions is to select appropriate classes in the object hierarchical model by traversing the tree from the root node to leaf nodes. However, there are cases when two or more nodes at the same level may be applicable for a design object. For example, in Figure 4-6, the class "y_comp_mem_strength_consideration" is divided into classes of "y_buckling_comp_mem" and "torsional_or_ftb_comp_mem" (ftb = flexural torsional buckling). The design member "column #125" must be checked for both buckling and torsional or flexural torsional buckling limit states. This thesis introduces both "AND" and "OR" relations in the Member Class Hierarchy, as shown in Figure 4-7. When traversing the AND-OR hierarchy, all nodes under an AND node are traversed, while only one node under an OR node is selected. Each OR node is linked to a classification method that classifies a design object into a more specific subclass. For the example shown in Figure 4-7, given a design member object "column #247" that is a steel non-slender compression member, the system traverses the object hierarchy in a depth-first manner as (using the alphabets as denoted in the figure):

A -> B -> D -> H -> K -> M -> P -> U -> K -> N -> R -> H -> L -> O

For each leaf node visited, the system stores the requirements linked to the leaf node classes. Thus, the design member object "column #247" is linked to the leaf nodes:

U, R, and O,

The requirements to be checked for the design object thus include:

- req_short_non_slender_y_comp_mem (linked to U)
- req_torsional_buckling_comp_mem (linked to R), and
- req_comp_mem_deflection (linked to O),

where the prefix "req_" denotes the requirement for a specific behavior limit state of the member type.

4.1.1.2 Classes

Each Class in the Member Class Hierarchy contains:

- property attributes, which correspond to the specific characteristics of the class object, and
- method attributes, which are pointers to the Method Objects representing the provisions.

Property attributes are used to describe the specific characteristics of the object class. There are two basic property attribute types:

- organizational attributes, which are used to represent the organization of the Member Class Hierarchy. They are:
  - super: its value indicates the super class,
  - and_or_node: its value indicates whether the node is an AND node or OR node,
  - and_sub: its value lists the child subclasses if the class is an AND node,
  - leaf: its value indicates whether the class is a leaf node,

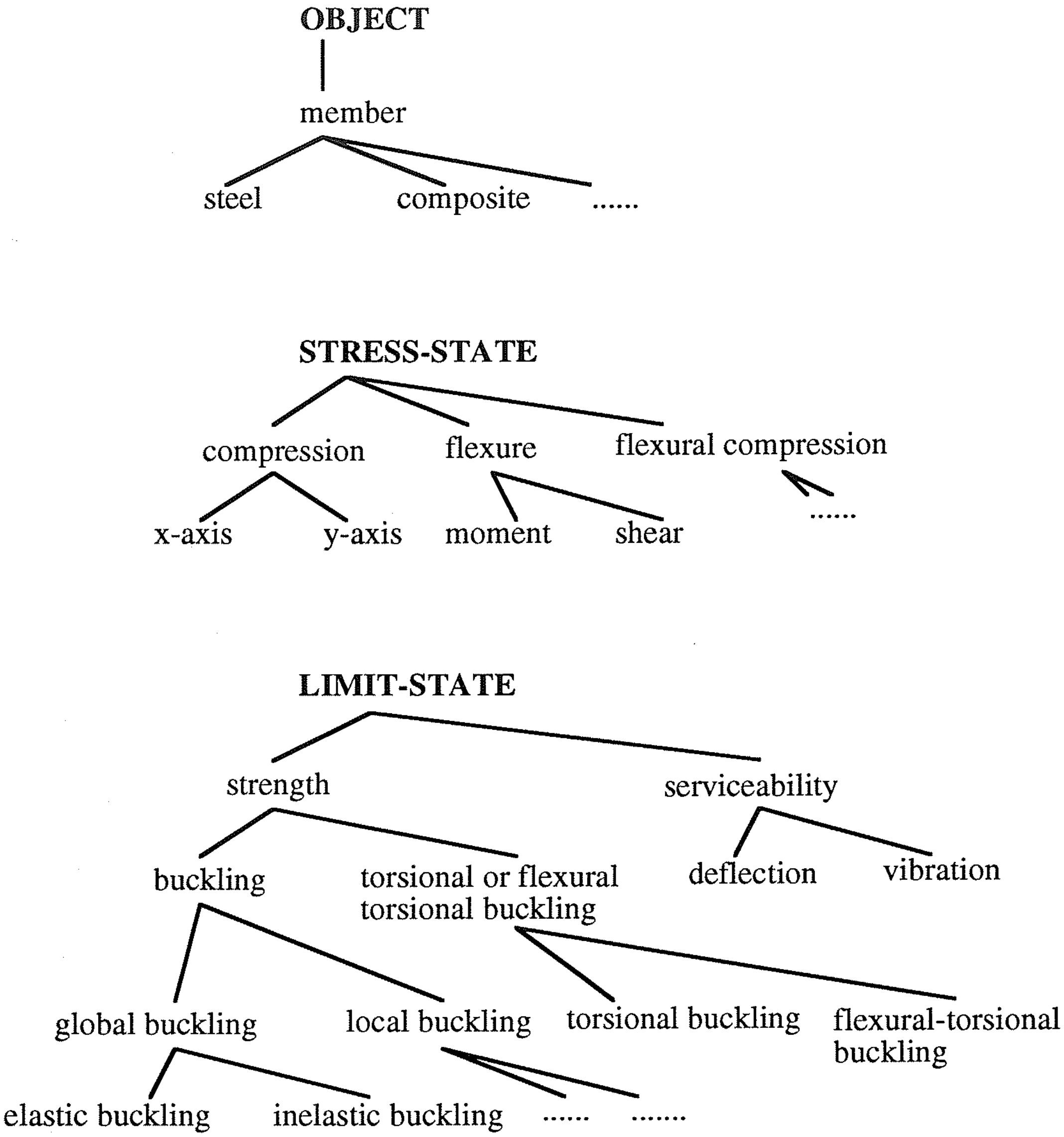


Figure 4-4 Fields and Partial Classifiers

  - requirement: its value contains the name of the requirement which the class is pointing to.
- object attributes, which are used to describe the basic properties of the objects such as "elastic_modulus," "yield_stress," and "unbraced_length_x."

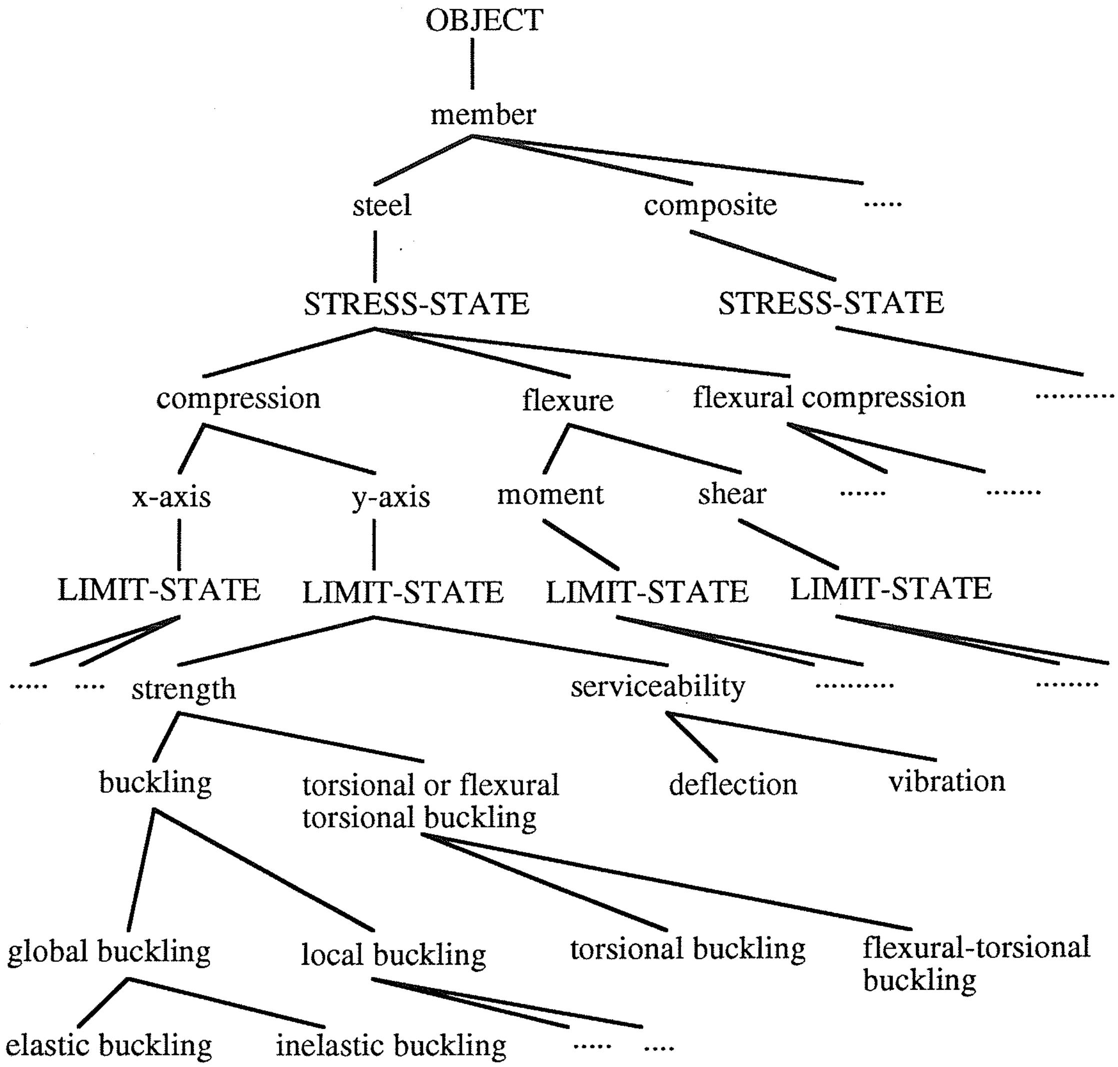


Figure 4-5 A Unified Hierarchical Tree

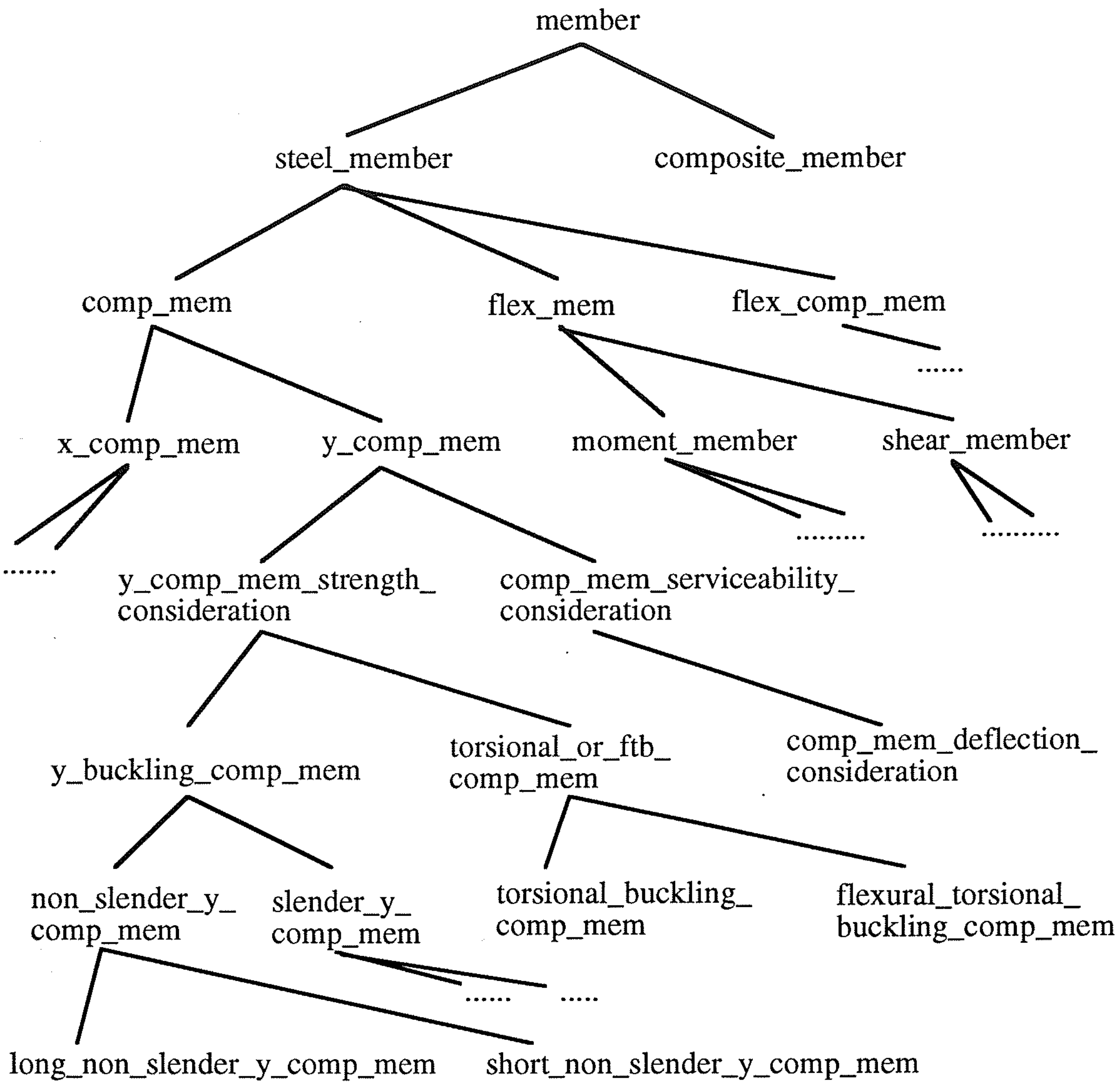

member
steel_member
composite_member
comp_mem
flex_mem
flex_comp_mem
......
x_comp_mem
y_comp_mem
moment_member
shear_member
....
....
....
y_comp_mem_strength_ consideration
comp_mem_serviceability_ consideration
y_buckling_comp_mem
torsional_or_ftb_ comp_mem
comp_mem_deflection_ consideration
non_slender_y_ comp_mem
slender_y_ comp_mem
torsional_buckling_ comp_mem
flexural_torsional_ buckling_comp_mem
......
.....
long_non_slender_y_comp_mem
short_non_slender_y_comp_mem

Figure 4-6 An Object-Oriented Unified Hierarchical Tree

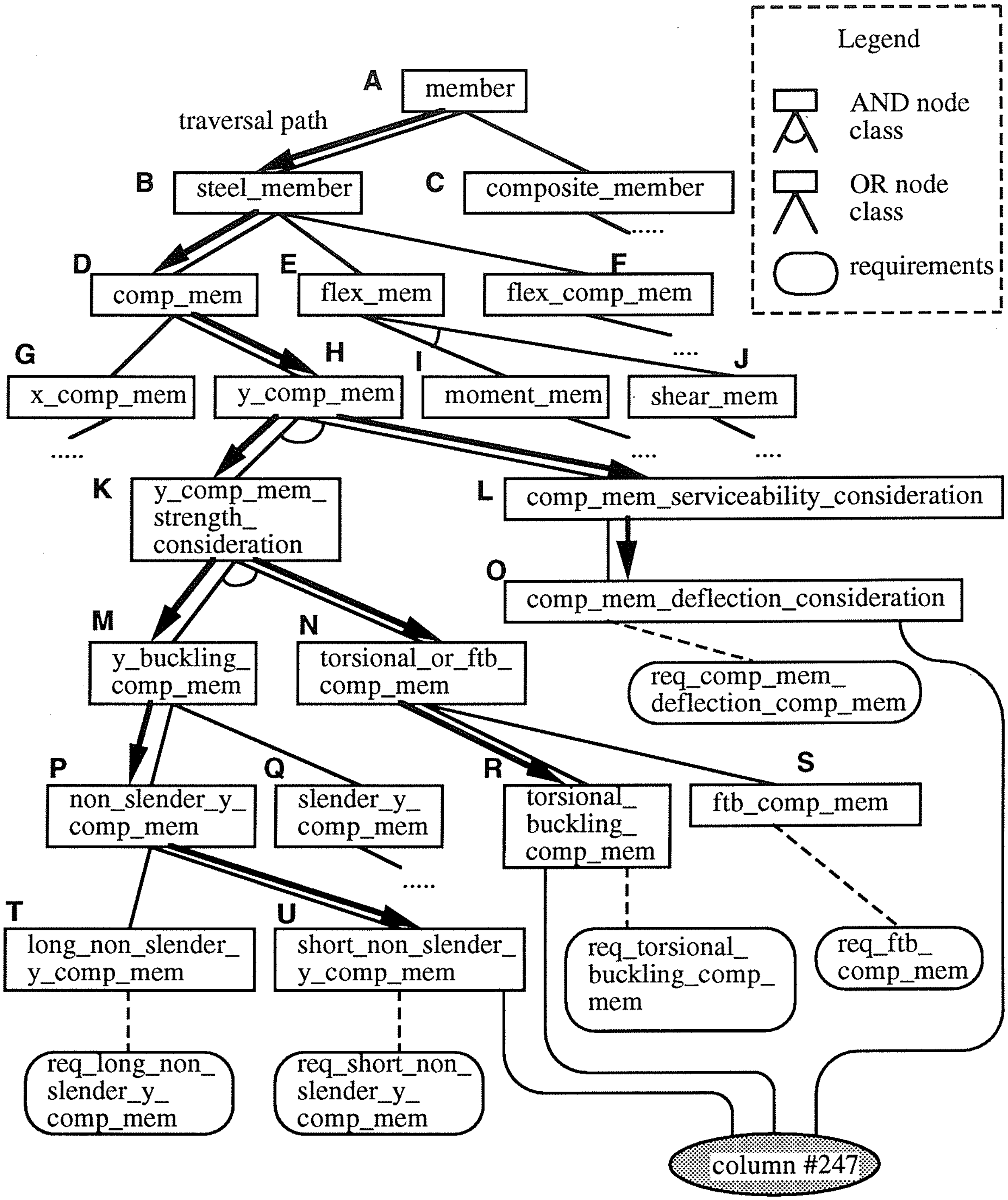

Legend
AND node class
OR node class
requirements
A member
traversal path
B steel_member
C composite_member
D comp_mem
E flex_mem
F flex_comp_mem
G x_comp_mem
H y_comp_mem
I moment_mem
J shear_mem
K y_comp_mem_ strength_ consideration
L comp_mem_serviceability_consideration
O comp_mem_deflection_consideration
req_comp_mem_ deflection_comp_mem
M y_buckling_ comp_mem
N torsional_or_ftb_ comp_mem
P non_slender_y_ comp_mem
Q slender_y_ comp_mem
R torsional_ buckling_ comp_mem
S ftb_comp_mem
T long_non_slender_ y_comp_mem
U short_non_slender_ y_comp_mem
req_torsional_ buckling_comp_ mem
req_ftb_ comp_mem
req_long_non_ slender_y_ comp_mem
req_short_non_ slender_y_ comp_mem
column #247

Figure 4-7 Object-Oriented Organization Model with AND-OR Tree

A method attribute consists of pointers that link an object class to its Method Objects representing specific provisions, i.e., requirements, determinants, and classifications in the provision. Figure 4-8 depicts the reference to the Method Objects by object classes in the Member Class Hierarchy via the method's attributes.

In the implementation of the model, the same attribute name of the requirements or determinants are used as the name of the Method Object. For example, the method attribute, its value of the requirement for torsional buckling for compression members, and the corresponding Method Object have the same name of:

req_torsional_buckling_comp_mem.

On the other hand, the method attribute for the classification method is always "classify." Both the value of the classification method of the class "torsional_or_ftb_comp_mem" and the corresponding classification Method Object have the name as:

clas_torsional_ftb_comp_mem.

That is, the method attribute provides a link between an object class in the Member Class Hierarchy and the corresponding Method Object.

4.1.2 Method Objects

Provisions of the design standard are represented in Method Objects. Each Method Object determines a single data item and is referenced by the method attribute of an object class in the Member Class Hierarchy. There are three types of Method Objects:

- requirements,
- determinants, and
- classifications.

A requirement is to check a given design situation and to deduce the conclusion of either "satisfied" or "violated." A determinant is a method to determine a data item on which other methods depend. At an OR node of the Member Class Hierarchy, a classification method is used to classify a design member object into a more specific subclass.

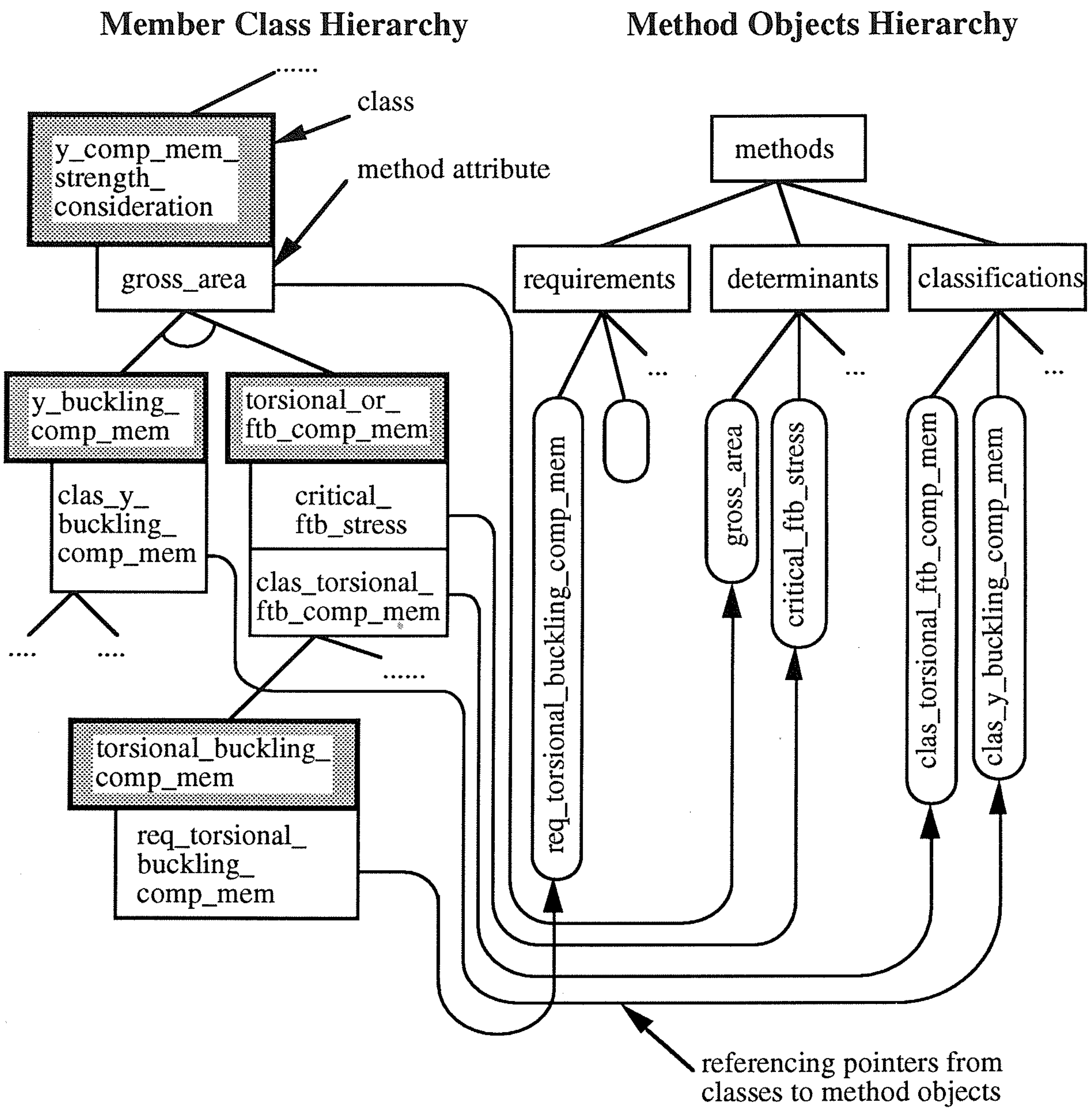

Member Class Hierarchy
Method Objects Hierarchy
......
class
method attribute
y_comp_mem_strength_consideration
gross_area
y_buckling_comp_mem
clas_y_buckling_comp_mem
torsional_or_ftb_comp_mem
critical_ftb_stress
clas_torsional_ftb_comp_mem
torsional_buckling_comp_mem
req_torsional_buckling_comp_mem
methods
requirements
determinants
classifications
req_torsional_buckling_comp_mem
gross_area
critical_ftb_stress
clas_torsional_ftb_comp_mem
clas_y_buckling_comp_mem
referencing pointers from classes to method objects

Figure 4-8 Member Class Hierarchy and Method Objects

The Method Objects are organized into a simple class hierarchy where the root node is "methods," as shown in Figure 4-8. The "methods" class has three subclasses, i.e., "requirements," "determinants," and "classifications." The "methods" class contains the general methods that any of its descendant classes and objects can inherit; for example, one generic method "input_attr" is implemented for requesting an input of an attribute value. Each requirement, determinant, or classification object is an instance of the class "requirements," "determinants," and "classifications" respectively. The "requirements" class contains general methods such as "respond_satisfied" and "respond_violated," which return the checked result of "satisfied" or "violated" respectively to the user that all requirement objects can inherit. Each Method Object has attributes and a method. The attributes defined in a Method Object consists of:

- identification: an indication that the method object is a requirement, a determinant, or a classification,
- class: an object class that points to the Method Object,
- provision: the provision number or section title in the design standard that the method represents, and
- reference: a list of determinant Method Objects or attributes that the Method Object itself references to determine the value of the Method Object.
- meaning: a brief explanation of the Method Object.

Methods are written in terms of Object-Logic sentences, which are based on a combination of object-oriented and logic programming. The key idea of logic programming is programming by description [Genesereth 85]. Because of their descriptive nature, provisions of design standards can be translated into logic sentences. Once the provisions are translated into logic sentences, conclusions can be deduced by the inferencing mechanism of resolution.

Despite the powerful inferencing capability, logic programming lacks basic features of software engineering like modularization, data abstraction, scoping, and information hiding [Page 89]. Thus, the object-oriented and logic programming paradigms are combined in the prototype development of the model.

In the Object-Logic model, methods are written in terms of Object-Logic sentences expressed as:

A :- C_1, C_2, ..., C_n.

or A.

A is a conclusion written in the form:

"*Name of Method*"(*Term*$_1$, *Term*$_2$, ..., *Term*$_m$)

where *Terms* are either variables, object constants, or functional expressions.

C_i (i = 1, 2, ... , n) is a condition which has the following structure:

Mem::"*Name of Method*" <- "*Message*,"

self <- "*Message*,"

or an arithmetic expression.

"*Name of Method*" is a method attribute value of the design member object variable "*Mem.*" "*Message*" is a literal which has the same form as the conclusion A. The symbol "<-" means that the "*Message*" is to be sent to the "*Method Object*" or "self"; and "self" denotes the Method Object itself. Each Method Object evaluates a single data item; that is, sentences that determine a particular data item are clustered in a Method Object. The Method Object contains only one method; that is, there is only one predicate for the conclusion part of the Object-Logic sentences defined in the Method Object. The predicate for the conclusion of Object-Logic sentences has the same named variable as the name of the Method Object.

In this model, each requirement is restricted to represent a single behavior of a member. If a provision has multiple requirements for different behaviors, the multiple requirement must be divided into basic individual requirements. Each basic requirement identifies a unique basic structural behavior constrained within the requirement. The Member Class Hierarchy is arranged in such a manner that each individual basic requirement is attached to a leaf node. This organization allows the user to identify the specific requirement that a design member violates. For example, the requirement for compression members with non-slender sections can be divided into four basic requirements according to the limit states of:

- inelastic or elastic buckling, and
- weak or strong axis buckling.

If the requirement for weak axis elastic buckling is violated, the user can decide whether to select a member with a larger radius of gyration about the weak axis or to provide an intermediate bracing along the weak axis of the compression member.

Each requirement Method Object contains at least two Object-Logic sentences corresponding to the values of "satisfied" and "violated." Figure 4-9 shows an example requirement method "req_short_non_slender_y_comp_mem." This method is linked to a leaf class "short_non_slender_y_comp_mem" (shown in Figure 4-7), which represents a compression member that is not slender and its inelastic member buckling is governed by the weak y-axis.

Let's examine the process of how a given design member object, say, "column_247" with the W shape section W14x99, satisfies a requirement. First, the message:

```
        req_short_non_slender_y_comp_mem(column_247, Result, w14x99)
```

is issued to the requirement Method Object. The Method Object receives the message and matches it by unifying the variables (Mem, Result, Id) with the values (column_247, satisfied, w14x99) respectively. The conditions in the Method Object are then being examined. In the first condition:

```
        Mem::pn_short_y <- pn_short_y(Mem, Pn, Id)
```

the system sends a message "pn_short_y(column_247, Pn, w14x99)" to a determinant Method Object "pn_short_y," as shown in Figure 4-10. The Method Object "pn_short_y" is a determinant which has a sentence:

```
        pn_short_y(Mem,Pn,Id):-
                Mem::gross_area<-gross_area(Mem, Ag, Id),
                Mem::fcry_short<-fcry_short(Mem, Fcry, Id),
                Pn is Ag * Fcry.
```

This sentence, in turn, sends a message "gross_area(column_247, Ag, w14x99)" to the Method Object "gross_area":

```
        gross_area(Mem, Ag, Id) :- Mem::area<-area(Id, Ag).
```

The Method Object "area" retrieves the data "Ag" from an appropriate database (which is to be described in the next section). Similarly, a message is sent to the Method Object "fcry_short" to evaluate the variable "Fcry." Once the variables "Ag" and "Fcry" are evaluated, the variable of "Pn" in the sentence "pn_short_y" is determined. The first condition of the requirement is then evaluated. Similarly, since the variable "Pn" has been determined, the second condition of the requirement,

```
        DS is 0.85 * Pn,
```

can be evaluated.

```
open_object req_short_non_slender_y_comp_mem.

  super = requirements.
  class = short_non_slender_y_comp_mem.
  provision = 'E2'.
  reference = [pn_short_y,
               attr(load_comp)].
  meaning = 'requirement for a non-slender compression
             member which y-axis inelastic buckling governs'.

  req_short_non_slender_y_comp_mem(Mem,satisfied,Id):-
      Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
      DS is 0.85 * Pn,
      self <- input_attr(Mem,load_comp,1000),
      Mem::eqless <- eqless(Mem::load_comp,DS),
      self <- respond_satisfied.

  req_short_non_slender_y_comp_mem(Mem,violated,Id):-
      Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
      DS is 0.85 * Pn,
      not(Mem::eqless <- eqless(Mem::load_comp,DS)),
      self <- respond_violated.

close_object req_short_non_slender_y_comp_mem.
```

Figure 4-9 An Example of a Requirement Method Object

```
open_object pn_short_y.

  super = determinants.
  class = short_non_slender_y_comp_mem.
  provision = 'E2'.
  reference = gross_area,
              fcry_short].
  meaning = 'nominal axial compression stress of a member which y-
             axis inelastic buckling governs'.

  pn_short_y(Mem,Pn,Id):-
      Mem::gross_area<-gross_area(Mem, Ag, Id),
      Mem::fcry_short<-fcry_short(Mem, Fcry, Id),
      Pn is Ag * Fcry.

close_object pn_short_y.
```

Figure 4-10 An Example of a Determinant Method Object

In the third condition of the requirement,

```
self<-input_attr(Mem, load_comp, 1000),
```

the method "input_attr" requests the user to enter the attribute value if it has not been assigned before. As noted earlier, this method is stored in the root class of the "methods" Method Objects hierarchy so that any Method Object can inherit and use this method. If the attribute value requested is available, this method simply returns true.

The fourth condition of the requirement,

```
Mem::eqless<-eqless(Mem::load_comp, DS),
```

is to check if the value of attribute "load_comp" is equal to or less than the value "DS." If it is true, the system proceeds to the last condition,

```
self<-respond_satisfied,
```

and informs the user that the requirement is satisfied; otherwise, the system backtracks this rule, then goes to the second rule. As noted earlier, the method "respond_satisfied" is located in the "requirements" class of the method object hierarchy which is shared by all requirement Method Objects.

A classification method is a procedure that classifies an OR node into more specific subclasses. For example, a compression member governed by y-axis member buckling with a non-slender section can be classified as "short" or "long" by the value of λ_{cy} in the AISC LRFD specification [AISC 86]. Let's examine how this classification method, shown in Figure 4-11, classifies a given design member "column_247." When the member comes to the class "non_slender_y_comp_mem" during the traversal of the Member Class Hierarchy, the message

```
classify(column_247,NextClass,w14x99)
```

is sent to the classification Method Object. Again, the variables (Mem, NextClass, Id) are unified with the constants (column_247, short_non_slender_y_comp_mem, w14x99). Then, the conditions in the first sentence,

```
Mem::lambda_c_y<-lambda_c_y(Mem, Lambda_c_y, Id),
Mem::eqless<-eqless(Lambda_c_y, 1.5),
writenl('classified as short_non_slender_y_comp_mem'),
```

are examined in the same manner with the requirement Method Object. In the first condition, the variable "Lambda_c_y" is determined. If this value is equal to or less than

```
open_object clas_non_slender_y_comp_mem.

  super = classifications.
  class = non_slender_y_comp_mem.
  provision = 'E2'.
  reference = lambda_c_y].

  classify(Mem,short_non_slender_y_comp_mem,Id):-
      Mem::lambda_c_y<-lambda_c_y(Mem, Lambda_c_y, Id),
      Mem::eqless<-eqless(Lambda_c_y, 1.5),
      writenl('classified as short_non_slender_y_comp_mem').

  classify(Mem,long_non_slender_y_comp_mem,Id):-
      Mem::lambda_c_y<-lambda_c_y(Mem, Lambda_c_y, Id),
      not(Mem::eqless<-eqless(Lambda_c_y, 1.5)),
      writenl('classified as long_non_slender_y_comp_mem').


close_object clas_non_slender_y_comp_mem.
```

Figure 4-11 An Example of a Classification Method Object

1.5 as noted in the second condition, the given member is classified as "short_non_slender_y_comp_mem," and the system writes that message on the computer screen. Otherwise, the second sentence is to be examined.

4.2 CAD Object Data Base

A member object to be designed or checked consists of attributes and external constraints. The attributes are the design properties of the member, such as effective length factor, unbraced length, and yield stress. External constraints are user requirements given to the design member object, such as the maximum depth of a beam. The CAD Object Data Base is used to define a design member object and to assign attribute values and external constraints to the design member.

The CAD Object Data Base consists of an Object Model, Data Objects, engineering databases, and a database (DB) Interface. Figure 4-12 shows the basic architecture of the CAD Object Data Base. The Object Model provides the user with a capability of defining a design member object, assigning attribute values and external constraints to the object.

Engineering databases contain data and information that are often not included in a design standard, such as dimensions and properties of standard component shapes, properties of various materials, and costs of materials, equipment, and labor. Data Objects and DB Interface facilitate retrieving the design data from an engineering database.

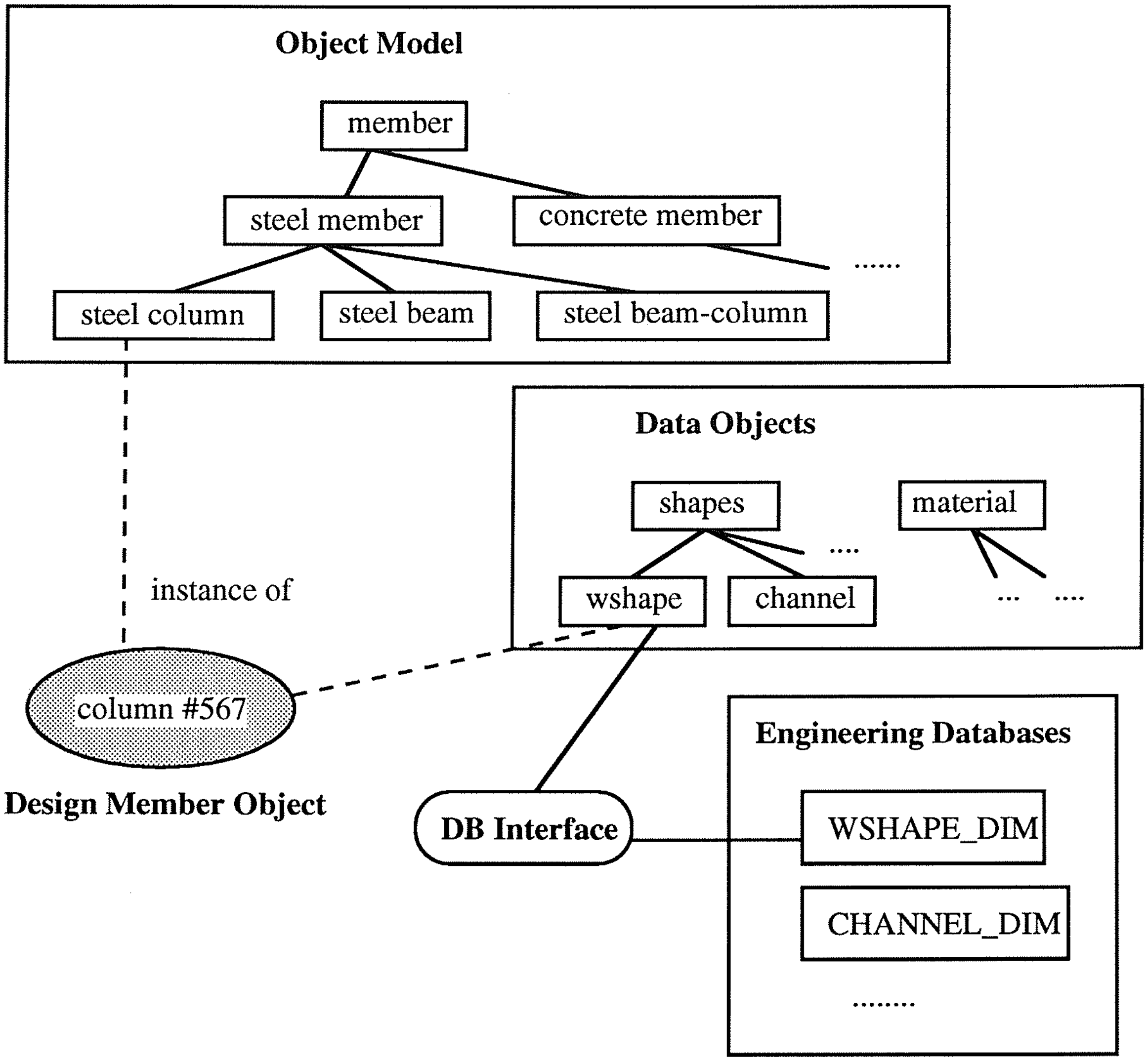


Figure 4-12 The CAD Object Data Base

4.2.1 Object Model

A design member object is a component in a physical facility structure, such as a beam, a column, or a truss member, to be designed or checked by the system. A design member object is created by the user as an instance of a class of the Object Model. The Object Model is an object hierarchy whose classes contain attributes. All the attributes within the Member Class Hierarchy are collected and assigned to appropriate classes of the Object Model. Figure 4-12 shows a sample design member object "column #567," linked to the class of "steel column" in the Object Model. The design member object "column #567" inherits all the attributes of the class "steel column" which contains all the attributes defined in the subtrees of "comp_mem" and "flex_comp_mem" of the Member Class Hierarchy because a column may be either a compression member or a flexural compression member.

Once the design member object is defined, the user can assign attribute values in a tabulated form provided by the system. The user may assign all, partial, or no attribute values prior to the execution of the design program. When the attribute value is needed during the execution of component design or conformance checking, the system requests the user to enter the value if it has not been given by the user earlier.

The user can assign external constraints to each design member object. External constraints to a design member object include architectural requirements such as restriction of the depth of a beam and serviceability constraints such as allowable deflection of a beam. External constraints are written in terms of Object-Logic sentences. The system provides several common types of external constraints so that users can directly modify such constraints to suit their needs. An example of external constraints restricting the weight of the component is shown below:

```
ext_constraints(Mem,satisfied,Id):-
        Mem::weight<-weight(Mem,Weight,Id),
        Weight < 15.
ext_constraints(Mem,violated,Id):-
        Mem::weight<-weight(Mem,Weight,Id),
        Weight >= 15.
```

These constraints are used and ensured for conformance in the design process.

Assigning attributes of the design member object is a time-consuming task. In a CAD environment, the attribute information about the design member object may be stored in a relational or an object-oriented database system. The Object Model provides a natural way to integrate the design system with a CAD object database system with an appropriate interface, which consists of mappings between the attributes of the Object Model and data of CAD object databases [Elam 88].

4.2.2 Engineering Databases and Data Objects

Engineering databases contain data such as dimensions and properties of standard component shapes, material properties, and costs of materials, equipments, and labor. Such data is usually necessary during conformance checking and component design, and is desired to be retrieved automatically by the system. The Object-Logic model naturally integrates itself with relational databases. Since relational calculus is a subset of logic, the uniformity of the relational and Object-Logic models can be retained.

Two approaches exist for coupling Object-Logic programs and relational databases: loose coupling and tight coupling. In the loose coupling approach, the data in the databases is loaded into the memory of Object-Logic programming environment as facts when the program is loaded. In the tight coupling approach, during the execution of the Object-Logic program, a query is sent to the database, and the results (tuples) are returned to the program [Ceri 90]. For conformance checking and component design, the tight coupling approach is better because the amount of engineering data is usually large and the interactions between the Object-Logic program and databases can be restricted to a small number of queries.

The interface between the Standards Base and engineering databases consists of a set of Data Objects and a DB Interface. Data Objects contain methods for data retrieval, written in terms of Object-Logic sentences. During the design process, the design member object is linked to appropriate Data Objects as their instance. The design member object can thus send query messages to the Data Objects to obtain necessary information from the engineering databases. The DB Interface couples Object-Logic programs and the relational database system, and is typically provided by commercial software suppliers, e.g., Quintus [Quintus 91].

Table 4-1 shows a relational table 'WSHAPE_DIM' storing the section dimensional properties of the W shapes. To retrieve the area of the W14x120 section, an SQL query is:

```
SELECT AREA
FROM WSHAPE
WHERE DESIGNATION = 'w14x120'.
```

In the Data Object "wshape," the query is expressed in the form of logic sentences as:

```
area(Designation,Area):-
  db_record('WSHAPE_DIM',[Designation,Area,_,_,_,_]).
```

This list [Designation,Area,_,_,_,_] in the querying sentences corresponds to the attributes of the table 'WSHAPE_DIM.' "_" in the list indicates an arbitrary variable. If a message

```
area(w14x120,Area)
```

is sent to the Data Object "wshape," the DB Interface then issues the query

```
db_record('WSHAPE_DIM',[w14x120,Area,_,_,_,_])
```

to the table 'WSHAPE_DIM' of the relational database system. The list in the query is unified with a list of values from the table and the variable "Area" is instantiated to have the value 35.3.

Table 4-1 A Part of a Relational Database 'WSHAPE_DIM'

Designation	Area	Depth	Web Thickness	Flange Width	Flange Thickness
....	....	....	....	....	....
w14x132	38.8	14.66	0.645	14.725	1.030
w14x120	35.3	14.48	0.590	14.670	0.940
w14x109	32.0	14.32	0.525	14.605	0.860
....	....	....	....	....	....

4.3 Summary

In this chapter, the Standards Base and the CAD Object Data Base of the Object-Logic system were described. In the Standards Base, the organization of the standard is represented by Member Class Hierarchy and the provisions are implemented as Object-Logic sentences in Method Objects. CAD Object Data Base consists of the Object Model, which provides the user with capabilities of defining a design member object, and engineering databases, which store the data that are not often included in the design standard. These data are retrieved during conformance checking or component design.

The advantages of the Member Class Hierarchy of the Object-Logic model are summarized as below. In the Object-Logic model, the unified Member Class Hierarchy is built from identified fields and classifiers, following an object-oriented structure. The "AND" relation is introduced to allow the design modules to identify all the applicable requirements systematically. A classification method attribute is attached to each OR node allowing the design member to traverse the class hierarchy properly.

In the Object-Logic model, attributes representing determinants and basic data items are attached to classes so that their lower classes inherit these attributes. This facilitates reorganizing and documenting the design standard. That is, design standards can be written based on the Member Class Hierarchy of the Object-Logic model, which contains all the attributes of requirements, determinants, and basic data items.

The method of developing the Member Class Hierarchy is generic and does not depend on design standards. Thus, multiple design standards, e.g., AISC LRFD and Allowable Stress Design (ASD) specifications [AISC 86] [AISC 89], can be included within a single Member Class Hierarchy [Garrett 86]. Method Objects are dependent of design standards. Thus, if multiple standards are included in the Standards Base, each Method Object needs a tag representing the design standard's name.

Chapter 5
The HyperDocument Model

This chapter describes in detail a HyperDocument model for the representation and documentation of design standards. HyperDocument is a generic document storage and retrieval model, following the current technology of HyperFile described in Section 2.2. HyperDocument model stores program codes representing provisions (Method Objects), standards provisions, their background information, and external programs as a set of documents.

Figure 5-1 depicts the overall system architecture of the HyperDocument model, which consists of a Document Base and a navigation system. The Document Base contains the electronic documents stored in the computer. Each document consists of its content and a HyperTag, which contains information about the document. A document can have pointers to reference other documents. The navigation system provides a set of generic facilities for document retrieval. The Document Base and the navigation system are presented in the following sections.

As shown in Figure 5-1, the HyperDocument model is separated from the rendering software, which is any kind of application software such as a word processing software, graphics software, spreadsheet, a CADD package, or multimedia software. The separation of rendering software from the HyperDocument model enables the user to select appropriate software for creating and rendering each document. As noted in Chapter 3, the HyperDocument model has been implemented in the prototype system HyperLRFD++. HyperTags and the Navigation system are implemented by HyperCard [Apple 90], and

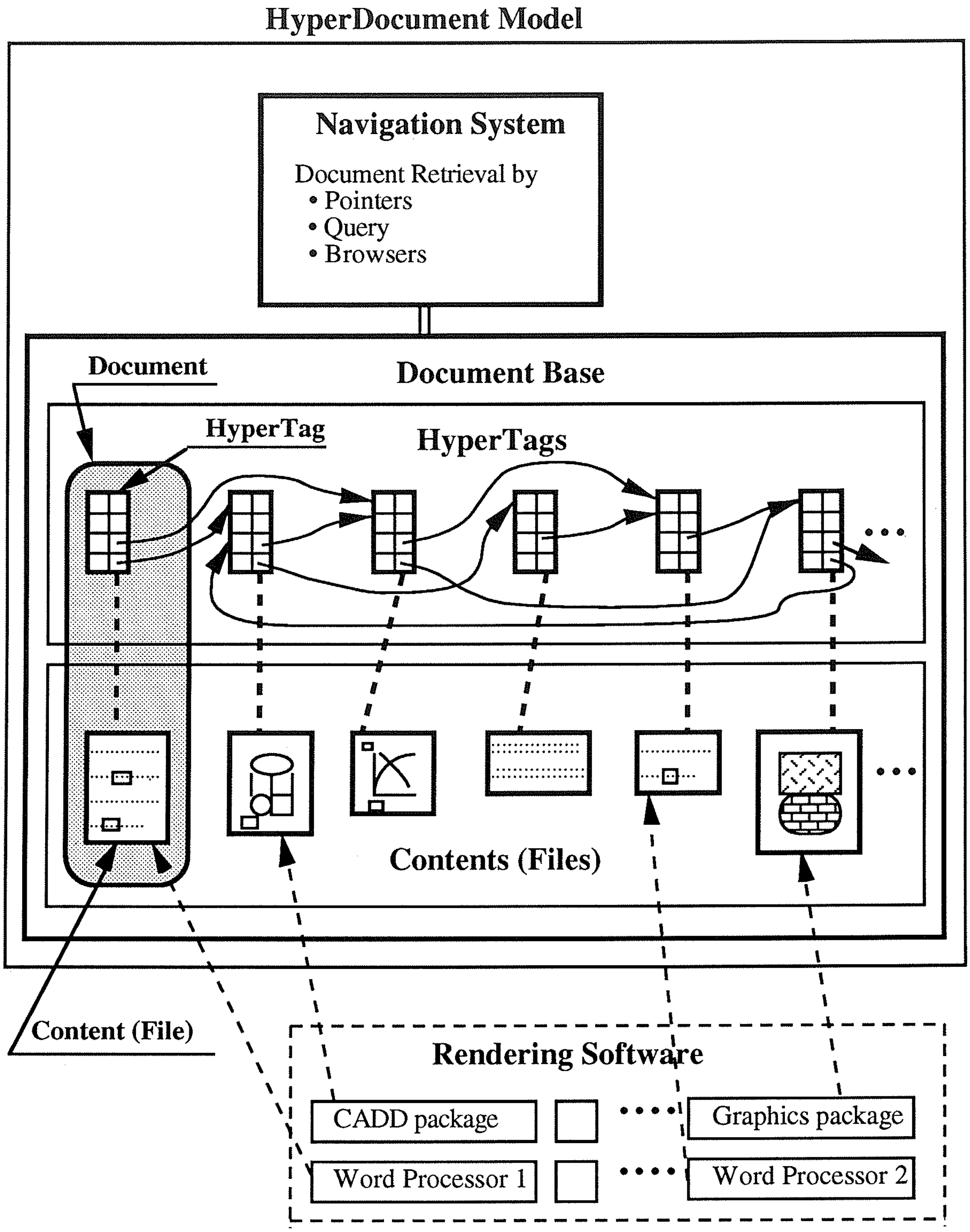


Figure 5-1 The Overview of the HyperDocument Model

contents are implemented by HyperCard, Prolog++ [Quintus 90], and Excel [Microsoft 91].

5.1 Document Base

This section describes the Document Base of the HyperDocument model. In the Document Base, a document consists of its content and HyperTag, as shown in Figure 5-2. HyperTag consists of attributes about the basic information of the document, such as its title, keywords, rendering software, content, and pointers. A number of pointers are set up to link the document to other documents, such as the referencing and the referenced documents, and the previous and next documents if they are in a sequence. The rendering software attribute specifies the application software needed to show the content of the document. The content attribute specifies a file name and its directory that stores the file of the document. Figure 5-2 shows an illustrative example of a document, where Section E2 of the AISC LRFD specification [AISC 86] is optically scanned and pasted on HyperCard [Apple 90].

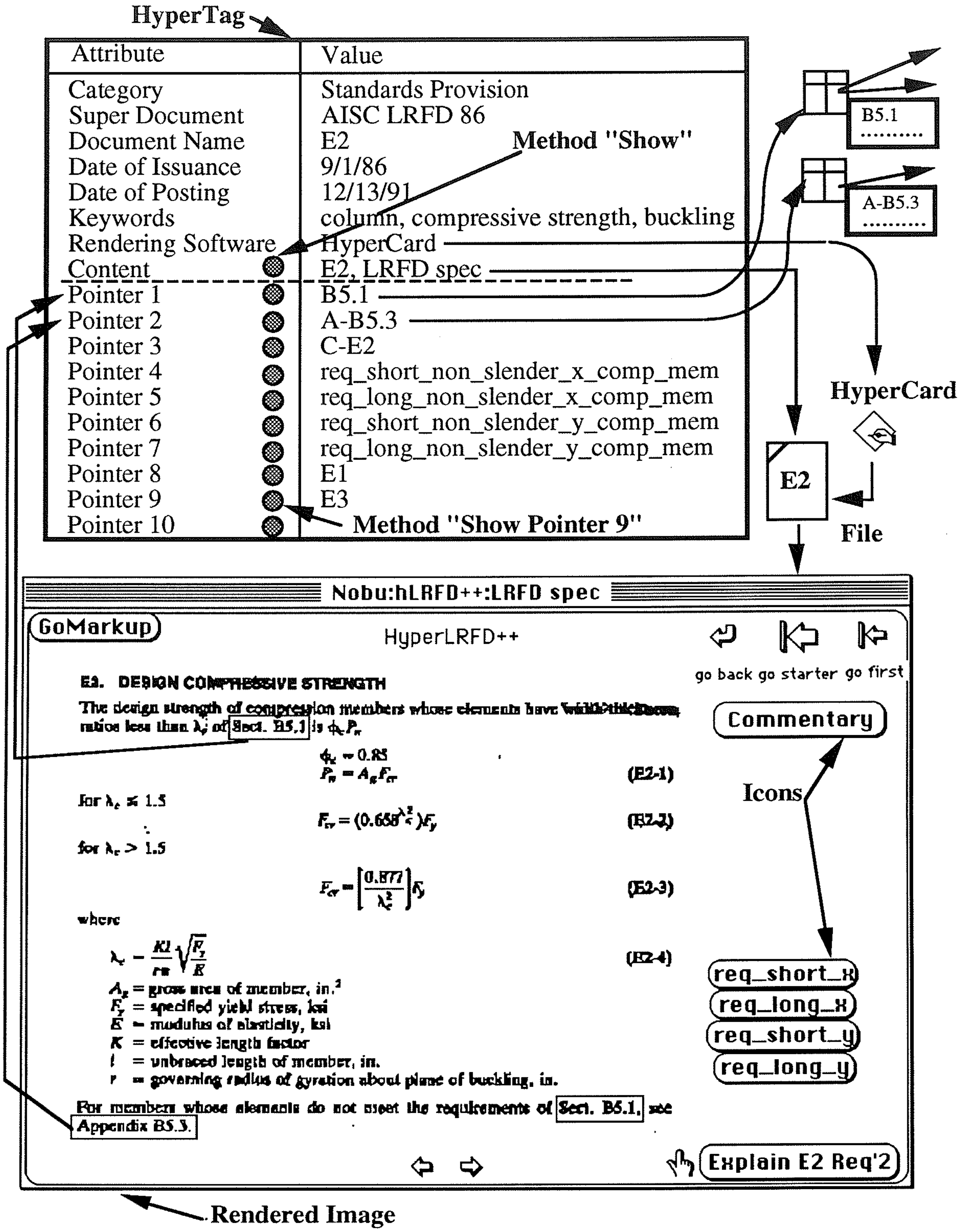
In the HyperDocument model for design standards, the Document Base consists of four document clusters:

- Method Objects (Provision Programs),
- Provision Document Base,
- Background Base, and
- External Programs.

Figure 5-3 shows a general structure of the Document Base. The four document clusters are described in the following subsections.

5.1.1 Method Objects

As described in Section 4.1.2, the Method Objects represent provisions as Object-Logic sentences. Method Objects are shared by both the Object-Logic model and the HyperDocument model. Method Objects are viewed as a group of executable programs in the Object-Logic model, and as a group of documents in the HyperDocument model. Each



Attribute	Value
Category	Standards Provision
Super Document	AISC LRFD 86
Document Name	E2
Date of Issuance	9/1/86
Date of Posting	12/13/91
Keywords	column, compressive strength, buckling
Rendering Software	HyperCard
Content ●	E2, LRFD spec
Pointer 1 ●	B5.1
Pointer 2 ●	A-B5.3
Pointer 3 ●	C-E2
Pointer 4 ●	req_short_non_slender_x_comp_mem
Pointer 5 ●	req_long_non_slender_x_comp_mem
Pointer 6 ●	req_short_non_slender_y_comp_mem
Pointer 7 ●	req_long_non_slender_y_comp_mem
Pointer 8 ●	E1
Pointer 9 ●	E3
Pointer 10 ●	

Nobu:hLRFD++:LRFD spec

GoMarkup

HyperLRFD++

go back go starter go first

Commentary

E2. DESIGN COMPRESSIVE STRENGTH

The design strength of compression members whose elements have width-thickness ratios less than λ_r of Sect. B5.1 is $\phi_c P_n$

$$\phi_c = 0.85$$

$$P_n = A_g F_{cr} \qquad \text{(E2-1)}$$

for $\lambda_c \leq 1.5$

$$F_{cr} = (0.658^{\lambda_c^2}) F_y \qquad \text{(E2-2)}$$

for $\lambda_c > 1.5$

$$F_{cr} = \left[\frac{0.877}{\lambda_c^2}\right] F_y \qquad \text{(E2-3)}$$

where

$$\lambda_c = \frac{Kl}{r\pi}\sqrt{\frac{F_y}{E}} \qquad \text{(E2-4)}$$

A_g = gross area of member, in.^{2}
F_y = specified yield stress, ksi
E = modulus of elasticity, ksi
K = effective length factor
l = unbraced length of member, in.
r = governing radius of gyration about plane of buckling, in.

For members whose elements do not meet the requirements of Sect. B5.1, see Appendix B5.3.

req_short_x
req_long_x
req_short_y
req_long_y

Explain E2 Req'2

Figure 5-2 A Sample Document in the Document Base

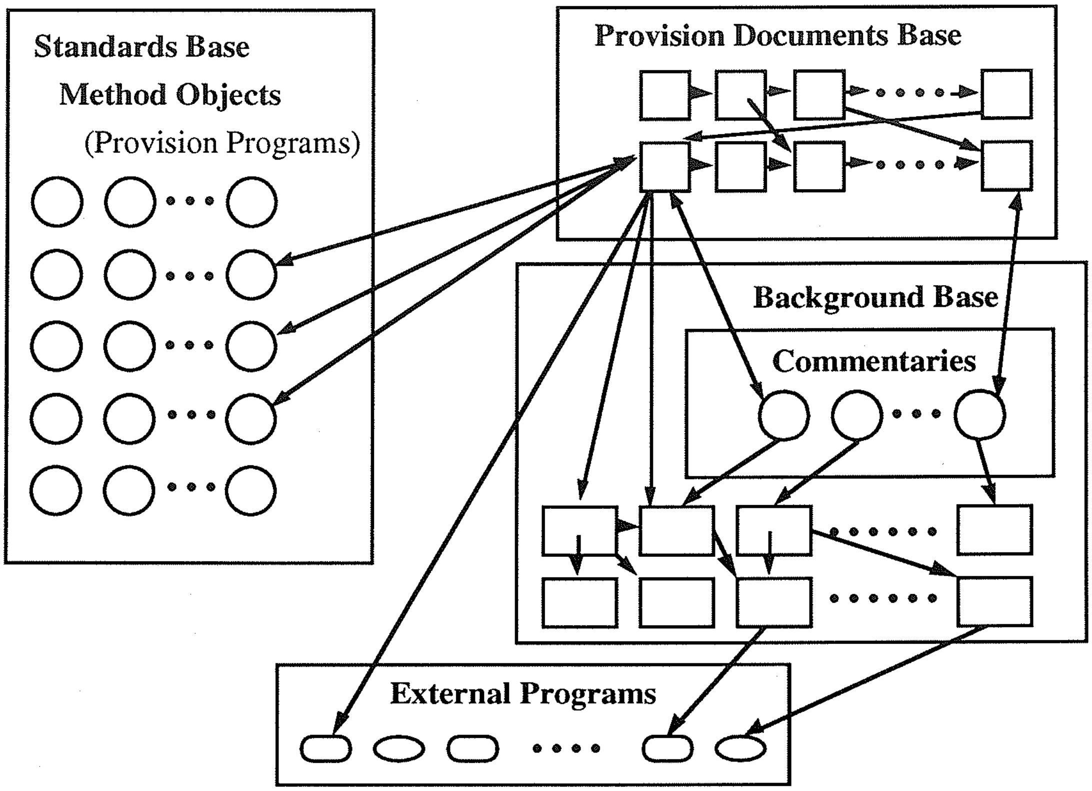


Figure 5-3 The Document Base of the HyperDocument Model

Method Object has a pointer to reference the corresponding provision document. Such references could be useful for the design and re-design of a member of a specific type.

5.1.2 The Provision Document Base

A standards provision is stored in the Document Base as text (with figures if any). In the provision document, pointers are set up to reference:

- Method Objects representing the provision,
- explanation documents about the provision,
- related Documents in the Background Base,
- external programs,
- referenced provision documents within the provision, and

- preceding and subsequent provision documents.

Such pointers are stored as attributes in the HyperTag.

If a provision consists of multiple requirements, each requirement is stored as a Method Object as discussed in Section 4.1.2. Thus, there is a one to many correspondence between the provision and its corresponding requirement Method Objects, i.e., one provision document has multiple pointers referencing the multiple requirement Method Objects while a requirement makes references to one provision document.

5.1.3 The Background Base

The Background Base contains explanations of the design provisions and various background information about them. The explanation documents include commentaries on the provisions and other important information such as historical notes and references. When creating or revising a standard, a large amount of information in various forms may be collected including:

- similar or related specifications,
- minutes of committee meetings,
- technical research reports and publications,
- experimental and measurement data,
- theoretical and computer solutions,
- video tapes of experiments, actual structural damages, etc.,
- audio tapes of experiments, interviews, etc.,
- graphics such as drawings, charts, maps, pictures, and photos, and
- miscellaneous resources.

Such background information can be processed by appropriate hardware and software, such as scanners and rendering software, and stored in a computer as a HyperDocument. Besides being the documentation of the standard development, the background information may be useful for design applications.

As discussed in Section 4.2, when the requirements are selected for conformance checking and used for component design, the design system reports the requirements that are violated. The user can now open documents about the violated provision and its background information that may be useful for re-design.

The Background Base provides a useful capability to store the knowledge, information, and data used for creating the provision. The documents can be shared among engineers and code developers and saved as a communication means when a design standard is revised. Engineers can also use the Background Base as an "electronic library" — a large document storage and retrieval system.

5.1.4 External Programs

During the design process, basic data items (input data) such as the yield stress and unbraced length of a compression member can be obtained easily from the user, CAD Object Databases, or Engineering Databases. However, many data items, such as the effective length factor *K* of a compression member in an unbraced frame, require more complex calculations and may include graphs and charts. External Programs may be needed to process charts, tables, and complex equations. These programs can be stored as part of the Document Base. While performing conformance checking and component designing in the Object-Logic model, the user can execute external programs to obtain such data items as effective length factor *K*. In this model, the procedure for executing an external program from the HyperDocument system is summarized as follows:

1. If a required data item is not readily available, the user can request to see the provision describing the data item.
2. If the user would like to see the description of the data item, the system shows the provision document; otherwise the user can simply enter the value.
3. If the provision document has a pointer to an external program that can be used to determine the data item, the user can execute the program by simply clicking the icon representing the program.
4. After processing the external program, the resulting value is passed to the Method Object of the Object-Logic model.

Figure 5-4 depicts the above procedure for the execution of External Program in the HyperDocument model.

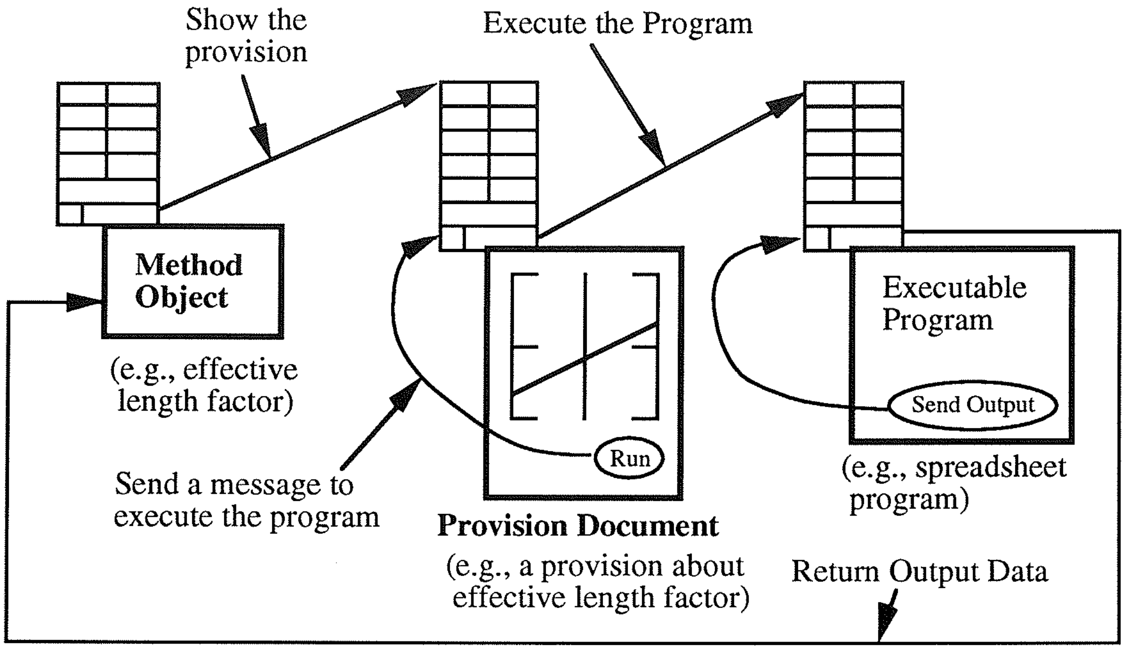

Show the provision
Execute the Program
Method Object
(e.g., effective length factor)
Send a message to execute the program
Run
Provision Document
(e.g., a provision about effective length factor)
Executable Program
Send Output
(e.g., spreadsheet program)
Return Output Data

Figure 5-4 Processing of the External Program

5.2 The Navigation System

Since the Document Base contains a large number of documents, the user needs an efficient means to access necessary documents. The Navigation system allows the user to retrieve documents from the Document Base easily and efficiently. There are three basic means to retrieve a document:

- Pointers or links,
- Queries, and
- Browsers.

They are described in the following subsections.

5.2.1 Navigation by Pointers

The user can navigate and retrieve a document by using the pointer attribute in the HyperTag. From a document, the user can traverse to the next document by simply clicking the corresponding method icon. This procedure can be repeated until the document being searched for is found.

The navigation system provides generic methods for traversing to referenced documents by pointers. A method "Show" shows the attribute <Content> of the HyperTag by opening the attribute <Rendering Software>. A method "Show Pointer 1" sends a "Show" message to the document linked by the attribute <Pointer 1>. When the document receives such a message, it invokes the "Show" method so that the content is rendered on the computer screen. Each "Show Pointer *N*" method (*N* = 1, 2, 3, ...) corresponds to <Pointer *N*> attribute. "Show" and "Show Pointer *N*" methods are represented as buttons and placed after <Content> and <Pointer *N*> attributes, respectively, as shown in Figure 5-2.

If the rendering software can include a button icon or a menu item that sends the method "Show Pointer *N*" of the HyperTag, the user can retrieve the linked document by clicking the icon or menu. Figure 5-2 shows a sample document, which is rendered by HyperCard [Apple90]. If the user clicks the rectangle icon embedded on the word "B5.1," the method "Show Pointer 1" represented as a button icon after the <Pointer 1> attribute is invoked and the document "B5.1" is rendered on the computer screen. If the document is rendered by software that cannot embed such icons and methods in the document file, the user can simply click the button on the HyperTag to retrieve the referenced documents.

5.2.2 Document Retrieval by Query

As used in database systems, querying is a powerful tool for document retrieval. In the HyperDocument model, the user can query by the attributes such as keywords and pointers. The navigation system provides three methods for querying the Document Base:

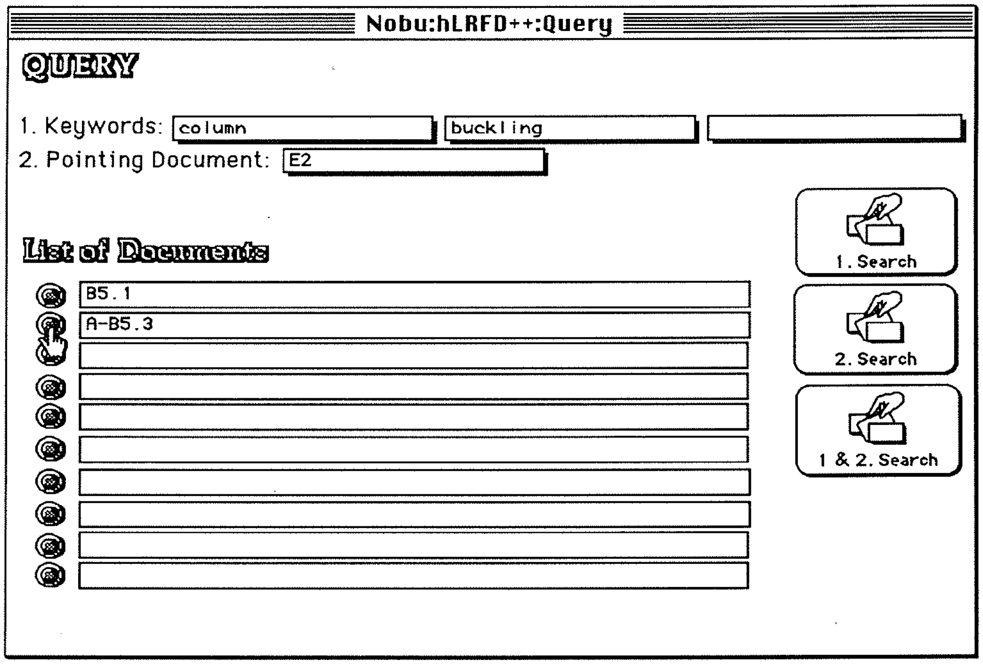
- Query by keywords:
  The user enters one to three keywords, then the processor searches documents containing the keywords. One example query "Show a list of document titles whose keywords contain 'column' and 'buckling'" is shown in the top portion of Figure 5-5.
- Query by pointers:

The user gives a document title, then the processor provides the titles of its linked documents. One example query "Show a list of document titles that the document 'E2' points to" is shown in the lower portion of Figure 5-5.

- Query by both keywords and pointers:

  The user can combine these two query methods by entering both keywords and pointing document title. The processor first enlists a set of linked documents, then selects documents that contain the given keywords. One example query "Show a list of document titles that the document 'E2' points to and whose keywords contain 'column' and 'buckling'" is shown in Figure 5-5.

The user interface includes a query menu and a document list board. The document list board shows a list of documents which are answers to the query and a set of button icons for the document list. If the user clicks the button, the corresponding document is shown. For example, if the user clicks the button placed at the left of the document title "A-B5.3," this document pops up on the screen as shown in Figure 5-6.



Nobu:hLRFD++:Query

QUERY

1. Keywords: column | buckling |

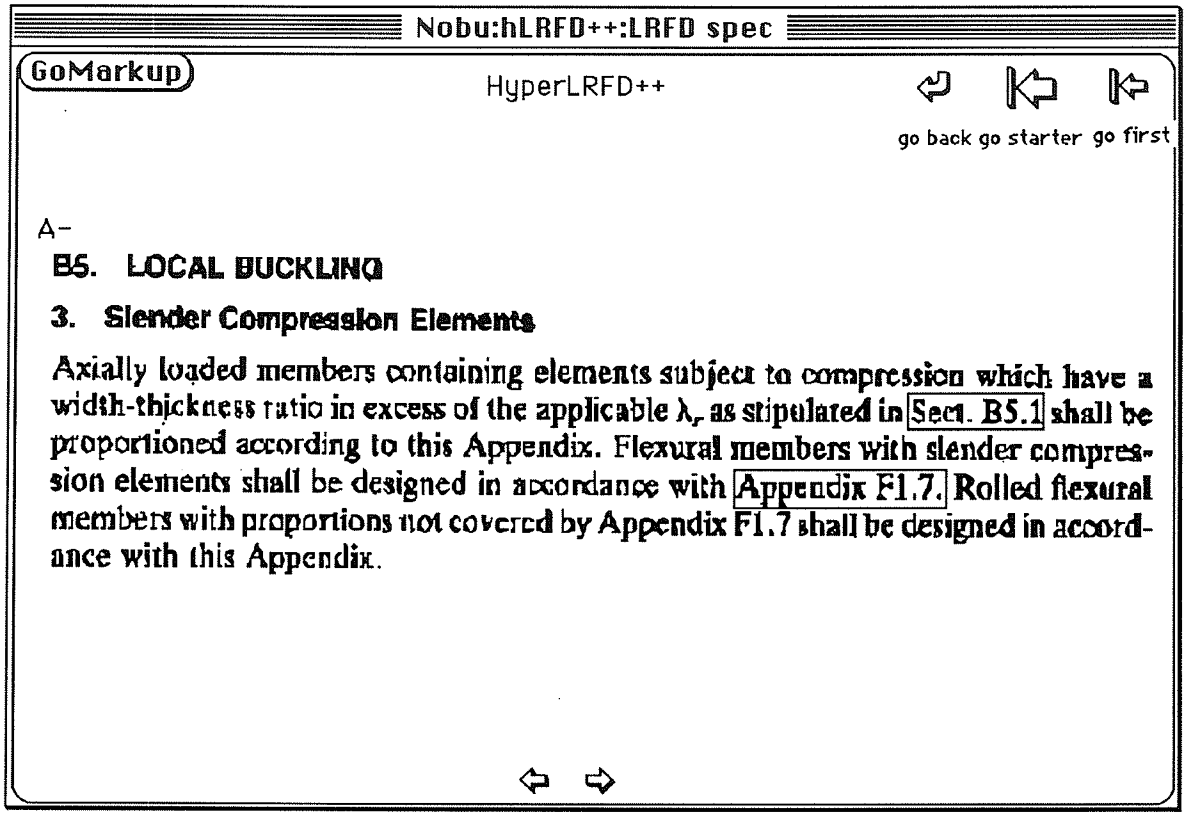
2. Pointing Document: E2

List of Documents

B5.1

A-B5.3

Figure 5-5 Query Menu and a Document List



A-

B5. LOCAL BUCKLING

3. Slender Compression Elements

Axially loaded members containing elements subject to compression which have a width-thickness ratio in excess of the applicable λ_r as stipulated in Sect. B5.1 shall be proportioned according to this Appendix. Flexural members with slender compression elements shall be designed in accordance with Appendix F1.7. Rolled flexural members with proportions not covered by Appendix F1.7 shall be designed in accordance with this Appendix.

Figure 5-6 The Document "A-B5.3" Retrieved by Query

5.2.3 Browsers

Another navigation tool is a browser. A browser allows the user to see a portion of the documents by a given context. Browsers of the HyperDocument model provide the user with an overview and history of visited documents to indicate the current location in the Document Base. An overview is a map of documents, which shows how documents are linked. Such an overview can be represented as a network where nodes represent documents' titles and links are pointers. An example of an overview network is shown in Figure 5-7. The overview network can be created by reading title and pointer attribute values of all the HyperTags in the Document Base. The history of visited documents is a list of visited documents' titles as shown in Figure 5-8. The user can read the document by clicking the button next to the document title. For example, if the user clicks the button

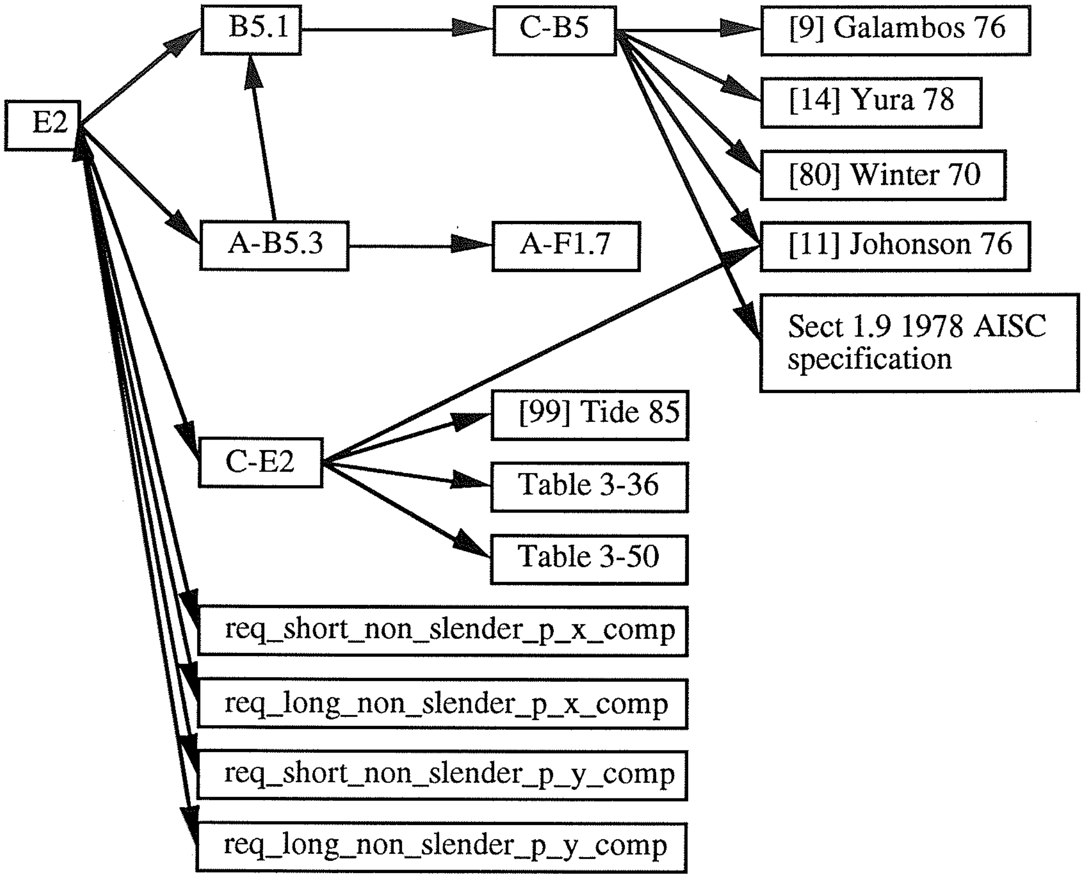
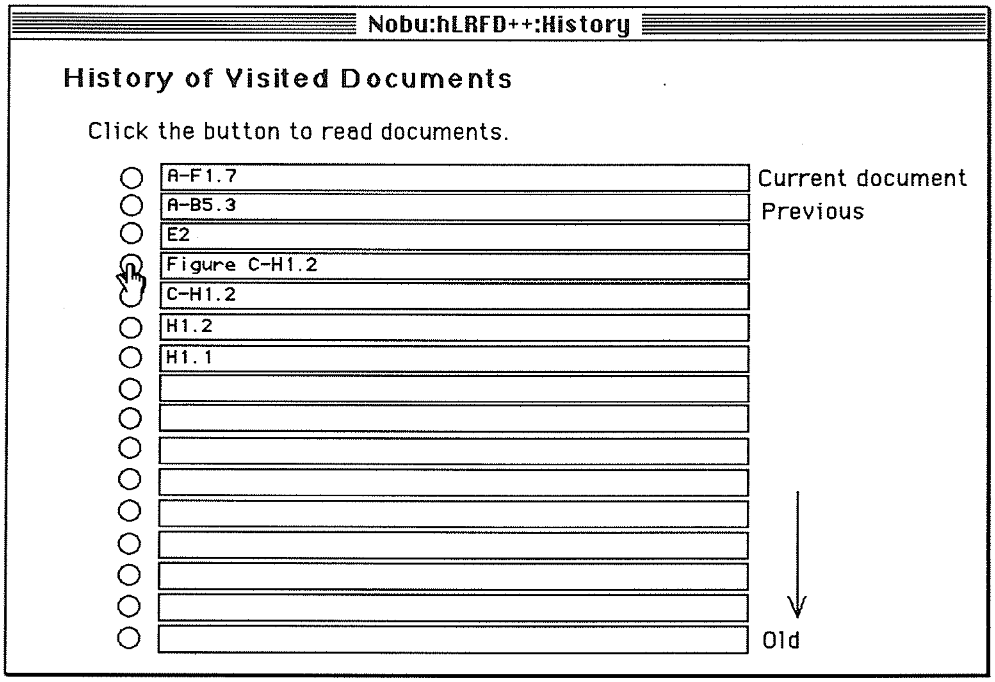

E2
B5.1
C-B5
[9] Galambos 76
[14] Yura 78
[80] Winter 70
[11] Johonson 76
Sect 1.9 1978 AISC specification
A-B5.3
A-F1.7
C-E2
[99] Tide 85
Table 3-36
Table 3-50
req_short_non_slender_p_x_comp
req_long_non_slender_p_x_comp
req_short_non_slender_p_y_comp
req_long_non_slender_p_y_comp

Figure 5-7 A Sample Overview Network

next to the document title "Figure C-H1.2," the system pops up the document shown in Figure 5-9.



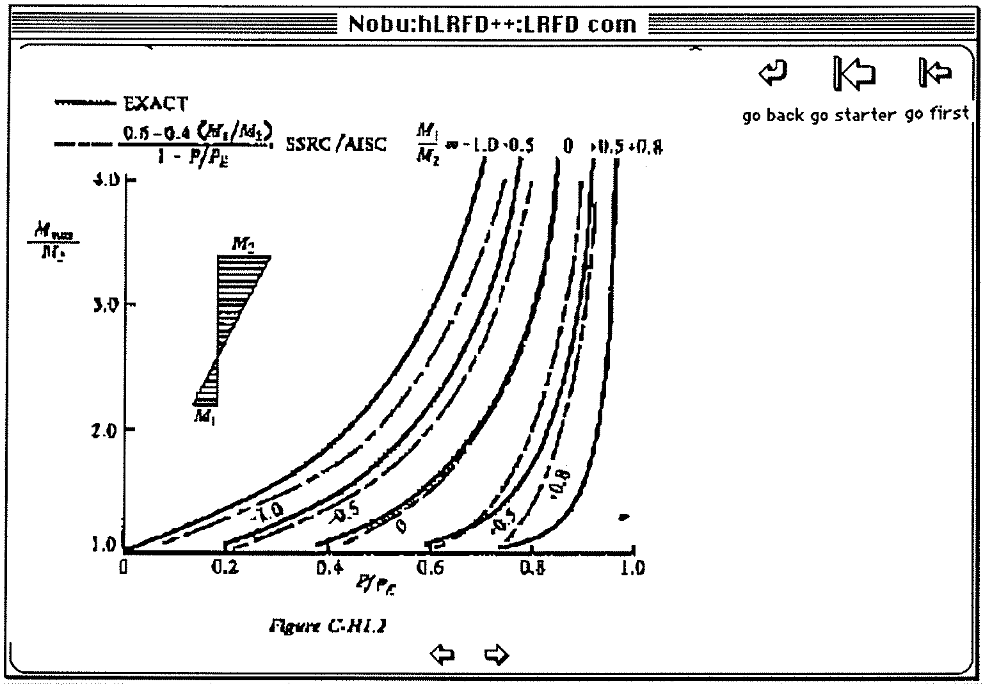
Nobu:hLRFD++:History

History of Visited Documents

Click the button to read documents.

○ A-F1.7 Current document
○ A-B5.3 Previous
○ E2
○ Figure C-H1.2
○ C-H1.2
○ H1.2
○ H1.1
○
○
○
○
○
○
○
○
○ Old

Figure 5-8 History of Visited Documents



Nobu:hLRFD++:LRFD com

go back go starter go first
EXACT
$\frac{0.6-0.4\,(M_1/M_2)}{1-P/P_E}$ SSRC/AISC
$\frac{M_1}{M_2}$ = -1.0 -0.5 0 +0.5 +0.8
$\frac{M_{max}}{M_2}$
4.0
3.0
2.0
1.0
M_2
M_1
-1.0
-0.5
0
+0.5
+0.8
0
0.2
0.4
0.6
0.8
1.0
P/P_E
Figure C-H1.2

Figure 5-9 The Document Retrieved by the Browser

5.3 Summary

This chapter described the HyperDocument model for the representation and documentation of design standards. The model is consists of the Document Base, which contains documents, and the Navigation system, which is used to create, delete, revise, and retrieve documents from the Document Base. The four document clusters in the Document Base, i.e., Method Objects, Provision Document Base, Background Base, and External Programs, were described. The three basic methods to retrieve a document, i.e., navigation by pointers, queries, and browsers, were presented. In this chapter preliminary investigation of the use of HyperFile technology for design standards processing and representation was discussed.

Chapter 6
Design Applications

In this chapter, the usage of the Hyper-Object-Logic model for design applications are illustrated. First, the Conformance Checking and Component Design Modules of the Object-Logic model are described. Four design application examples of the prototype system, HyperLRFD++, are illustrated. Section 6.1 describes the Conformance Checking and Component Design Modules. Section 6.2 illustrates the preliminary design of a compression member. Section 6.3 presents a conformance checking of the compression member as designed in Section 6.2. Section 6.4 presents conformance checking of a flexural member. In Section 6.5, the detailed design of a flexural compression member is presented. Section 6.6 summarizes this chapter.

6.1 Conformance Checking and Component Design Modules

6.1.1 Conformance Checking Module

Conformance checking refers to the evaluation of a designed member whether it satisfies all the applicable requirements as defined in a design standard. For detailed design, the system identifies all the applicable requirements and automatically executes these requirements based on the given attribute values and external constraints of the member.

For preliminary design, however, the user can focus on checking a single or limited number of specific requirement(s) for a specific object type, stress state, or limit state by selecting a specific class in the Member Class Hierarchy. This procedure reduces the system's execution time and interactions between the user and the system. That is, the user can prune the unnecessary or unimportant requirements that are not of primary concerns in the preliminary design of the member.

The conformance checking procedure consists of six basic steps:

1. The user defines a design member object.
2. The system links the design member object to engineering databases through the Data Objects and the DB Interface.
3. The user selects a design strategy, that is either preliminary or detailed design.
4. The system traverses the Member Class Hierarchy and records all the applicable requirements.
5. The system executes the applicable requirements.
6. The system reports the result to the user.

The operations are described in detail in the following paragraphs.

For conformance checking, the user first defines the design member object and provides the data about the object. The user can define the object as an instance of a class in the Object Model and type in attribute values for the design member to be checked. The user can give either all, partial, or no attribute values about the design member. If an attribute value is not available during the execution of conformance checking, the system would request the user to enter the value or to accept the default value provided by the system. In addition, the user can impose external constraints to the design member object, as discussed in Section 4.2.1.

The system links the design member object to appropriate engineering databases via Data Objects and DB Interface. The Data Object becomes a parent to the design member object. As noted in Section 4.2.2, the data about the design member can be retrieved from an engineering database.

As noted earlier, the user can check the design member object for a preliminary or detailed design. For preliminary design, the user also selects a specific class in the Member Class Hierarchy that the member object is to be checked. The system first checks whether or not the user-selected class is appropriate for the designated member by backtracking the Member Class Hierarchy from the selected class. For example, if the user selects the class "non_slender_y_comp_mem" in Figure 6-1, the member "column #247" backtracks from this class to the root node "member." During this process, the system executes the classification method of the class which is the parent of the class node where "column #247" is temporarily located. If the class is not appropriate for the given design member, the system warns the user and asks the user whether the execution should continue. The user has full control on design for a particular requirement and continues the session if it is so desired.

If the user selects the preliminary design and the selected class is affirmed, its subclasses of the Member Class Hierarchy are traversed to determine the appropriate requirements. In the example shown in Figure 6-1, the member "column #247" traverses only the nodes N and S, and the design member object is linked to the leaf node S "short_non_slender_y_ comp_mem." If the user selects the detailed design option, the entire Member Class Hierarchy is traversed and applicable requirements are identified and checked. As the Member Class Hierarchy is traversed, the system records all the applicable requirements linked to the leaf classes traversed by looking at their requirement attributes. When the traversal is completed, a list of all the applicable requirements is recorded.

For each applicable requirement and its corresponding Method Object, the system checks for conformance based on the data about the design member. As discussed in Section 4.1.2, the system sends a message to each requirement Method Object to determine whether the requirement is satisfied or violated. Once the message is received by the Method Object, the conditions of the rules in the object are examined by the resolution principle and objects' message passing.

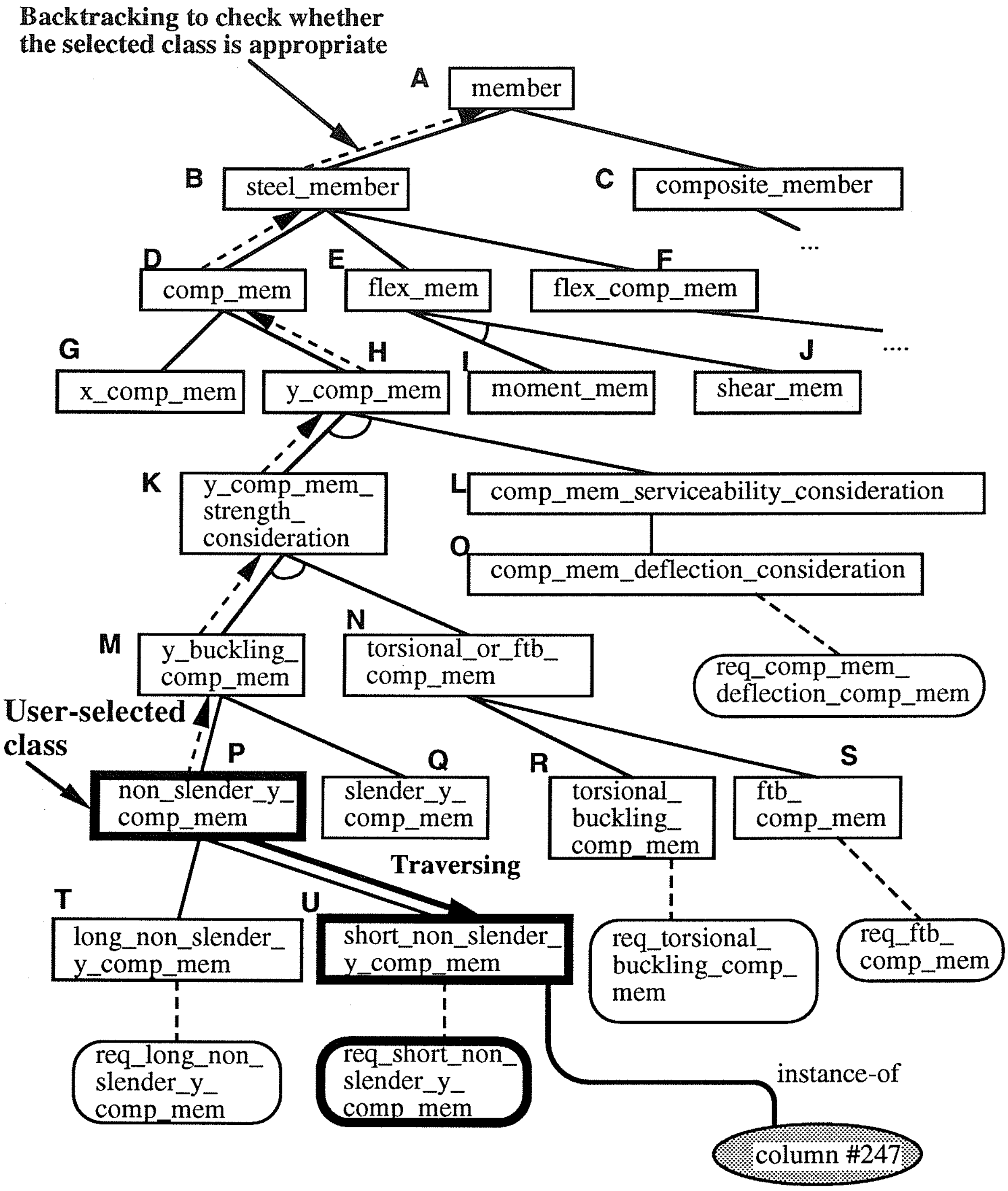
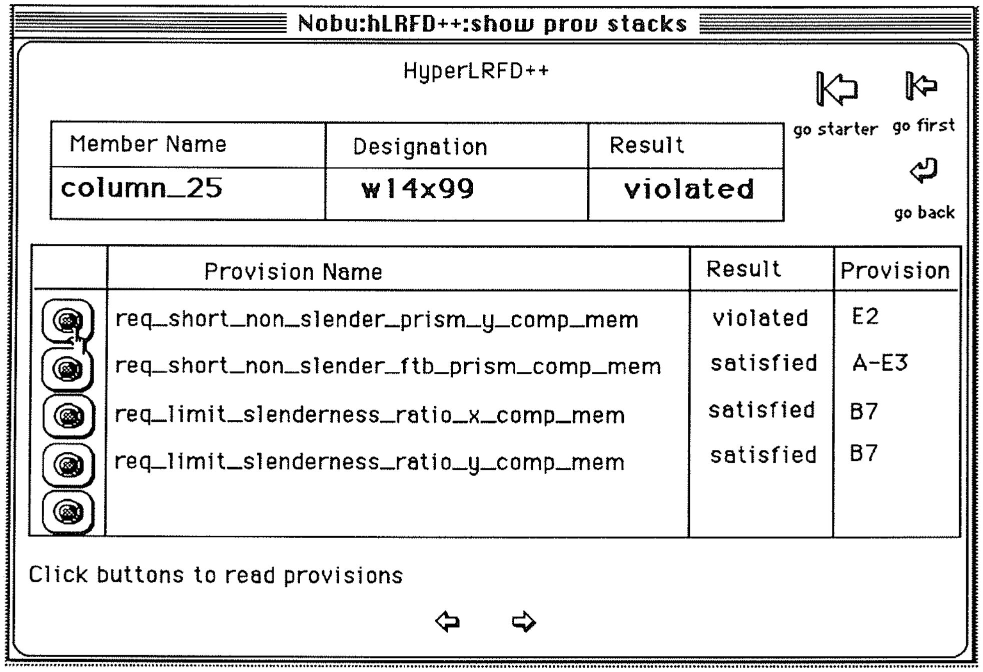

Backtracking to check whether the selected class is appropriate
A member
B steel_member
C composite_member
...
D comp_mem
E flex_mem
F flex_comp_mem
....
G x_comp_mem
H y_comp_mem
I moment_mem
J shear_mem
K y_comp_mem_ strength_ consideration
L comp_mem_serviceability_consideration
O comp_mem_deflection_consideration
M y_buckling_ comp_mem
N torsional_or_ftb_ comp_mem
req_comp_mem_ deflection_comp_mem
User-selected class
P non_slender_y_ comp_mem
Q slender_y_ comp_mem
R torsional_ buckling_ comp_mem
S ftb_ comp_mem
Traversing
T long_non_slender_ y_comp_mem
U short_non_slender_ y_comp_mem
req_torsional_ buckling_comp_ mem
req_ftb_ comp_mem
req_long_non_ slender_y_ comp_mem
req_short_non_ slender_y_ comp_mem
instance-of
column #247

Figure 6-1 Requirement Focusing in the Preliminary Design Phase



Nobu:hLRFD++:show prov stacks

HyperLRFD++

Member Name	Designation	Result
column_25	w14x99	violated

	Provision Name	Result	Provision
◉	req_short_non_slender_prism_y_comp_mem	violated	E2
◉	req_short_non_slender_ftb_prism_comp_mem	satisfied	A-E3
◉	req_limit_slenderness_ratio_x_comp_mem	satisfied	B7
◉	req_limit_slenderness_ratio_y_comp_mem	satisfied	B7
◉			

Click buttons to read provisions

Figure 6-2 An Example of a Conformance Checking Result

When all the requirements are checked, the results are presented in a tabulated form as shown in Figure 6-2. The information presented includes:

- Name of the design member object,
- Design ID such as a W shape designation (e.g., w14x99),
- Result (satisfied or violated) for each requirement,
- Requirements and their icons such as buttons, and
- Provision title for each requirement.

If the member violates a requirement, the user can read the corresponding provision and its explanation stored in the HyperDocument system by clicking the provision button icons on the result table. This information can be useful when re-design is necessary. For example, suppose a beam violates a requirement for the shear stress of a thin web but satisfy all other requirements. The user may want to select a section with a thicker web without changing other parts of the design.

6.1.2 Component Design Module

In the Object-Logic model, the design approach is to generate a plausible component design using a set of heuristics and to test the design for conformance checking. The design heuristics are procedures applicable for a group of members governed by appropriate requirements and are expressed in Object-Logic sentences. For example, for selecting a least weight W shape section as a flexural member, one heuristic is based on the W shapes listed according to the plastic section Module Z_x, as given in Part 3 of LRFD Manual [AISC 86]. A partial example list is:

```
[........,[w10x54,w14x43,w12x50,w16x40],
          [w10x60,w12x53,w14x48,w18x40],
          [w16x45,w12x58], .........................]
```

The list is subdivided into groups such as [w16x45, w12x58] in such a manner that the last W shape designation of each group is the least weight section within the group. For each shape designation in the list, we can compute the maximum design resisting moment:

$$\phi_b M_p = \phi_b Z_x F_y \,/\, 12$$

where $\phi_b = 0.85$, M_p = plastic bending moment, F_y = yield stress. If a section (e.g., w12x53) that $\phi_b M_p$ equals or exceeds the required strength is found, the last designation of the section's group is the plausible design (e.g., w18x40). Since this design heuristics is based on the requirement for a compact flexural member whose unbraced length is shorter than the limiting laterally unbraced length for full plastic bending capacity L_p, the selected section is to be checked for conformance of the assumptions as noted in the member class. If the plausible design violates other requirements or external constraints, the next least weight designation is to be selected as a candidate design (e.g., w12x58).

The procedure for component design can be summarized as follows:

1. The user defines a design member object and selects a focused object class deemed most appropriate for the design member object.
2. The system generates a component based on the heuristics derived from the requirement of the focused object class.
3. The system checks the design for conformance with the applicable requirements.

4. If the design violates a requirement or external constraints, the system generates another plausible design and return to step (3). If the design conforms with all the requirements and the user is satisfied with the design, proceed to step (5).
5. The system generates a design report.

The following sections present conformance checking and component design examples.

6.2 Preliminary Design of a Compression Member

This section describes an example for the preliminary design of a column (Figure 6-3) using the prototype system HyperLRFD++. While both the column and the beams are assumed to be W shapes, their designations are not given. There are four basic steps in the preliminary design process:

1. Problem definition: the user defines the column and the known attribute values.
2. Initial design: the system selects a trial W shape designation.
3. Design checking: the system checks the conformance of the trial section against the design provisions, and selects a new trial section if the design provisions are violated.
4. Post processing: the system summarizes and presents the result to the user.

In the following subsections, this preliminary design procedure is described in detail.

6.2.1 Problem Definition

There are three basic steps that the user follows in defining the column design problem:

- opening the HyperLRFD++ interface,
- defining the member object as an instance of the "steel column" class of the Object Data Model, and
- assigning attribute values of the member.

These tasks are described in this section.

To begin the preliminary design of the column, the user first opens the user interface of HyperLRFD++, as shown in Figure 6-4. The user interface consists of a HyperCard stack that provides instructions and facilities for the design procedures. The user opens Oracle, a relational database system, which stores the properties of W shape sections, by clicking the first button.

To assign known attribute values to the member, the user clicks the second button for "Define Member." The user defines the member, "column_25," as an instance of a class "steel column" in the Object Model shown in Figure 6-5. "Column_25" inherits attributes from the class "steel column." Note the Object Data Model is in the CAD Object Data Base and is not the same as the Member Class Hierarchy in the Standards Base.

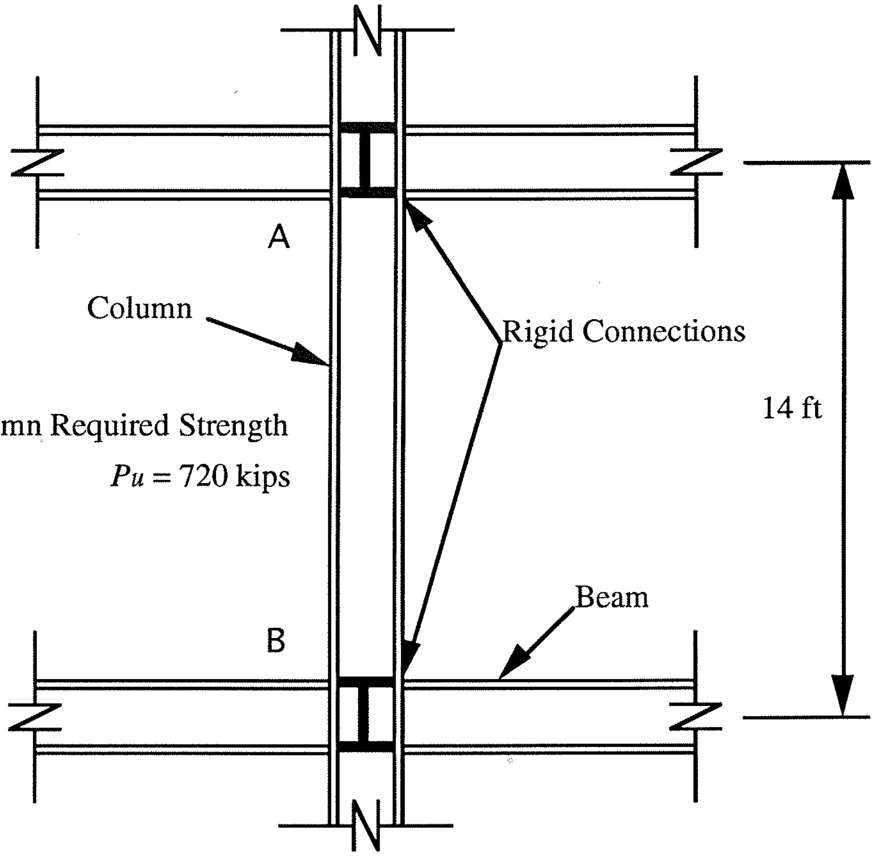
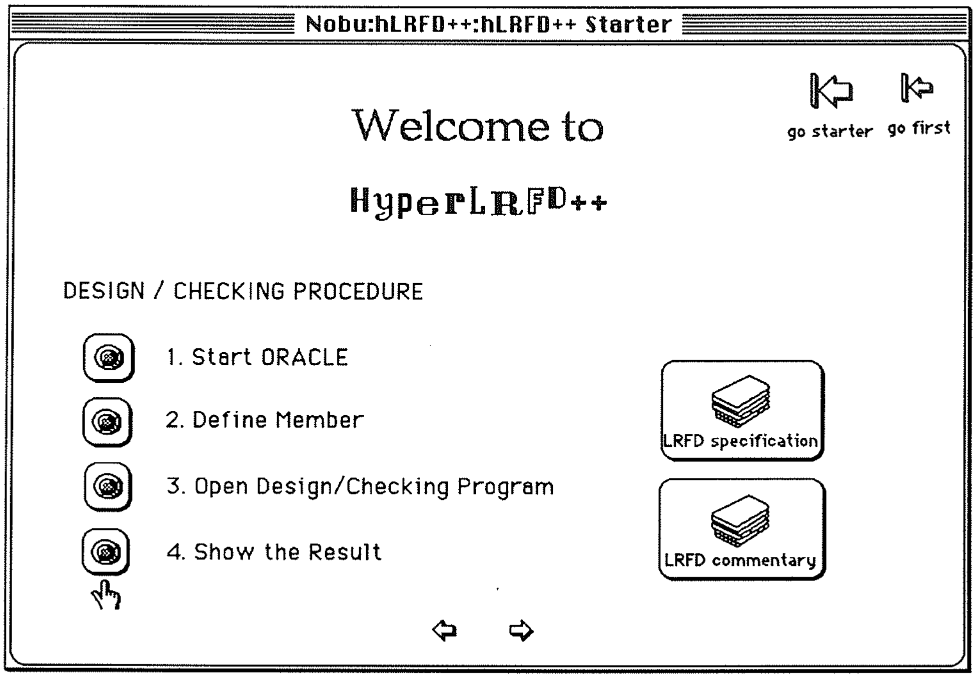


Figure 6-3 An Example for Column Design (Adapted from [Rokach 91])



Nobu:hLRFD++:hLRFD++ Starter

Welcome to

HyperLRFD++

DESIGN / CHECKING PROCEDURE

1. Start ORACLE

2. Define Member

3. Open Design/Checking Program

4. Show the Result

Figure 6-4 The First Card of the User Interface

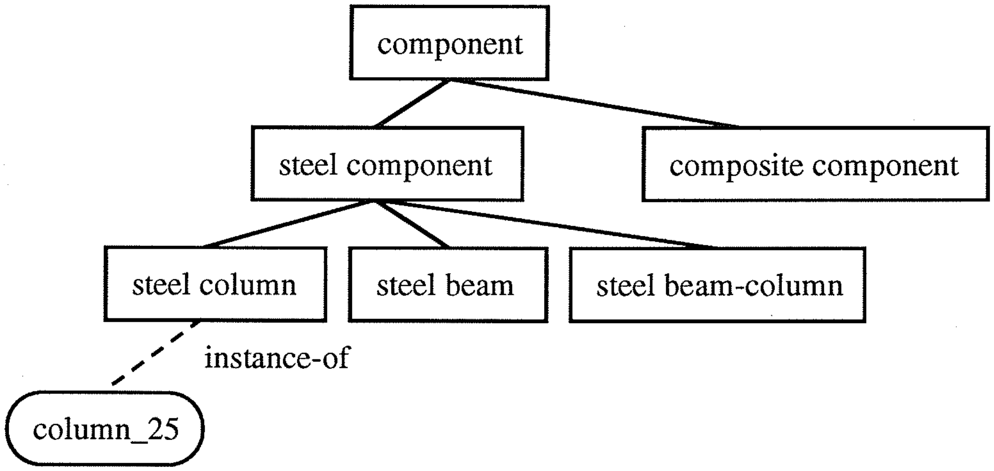
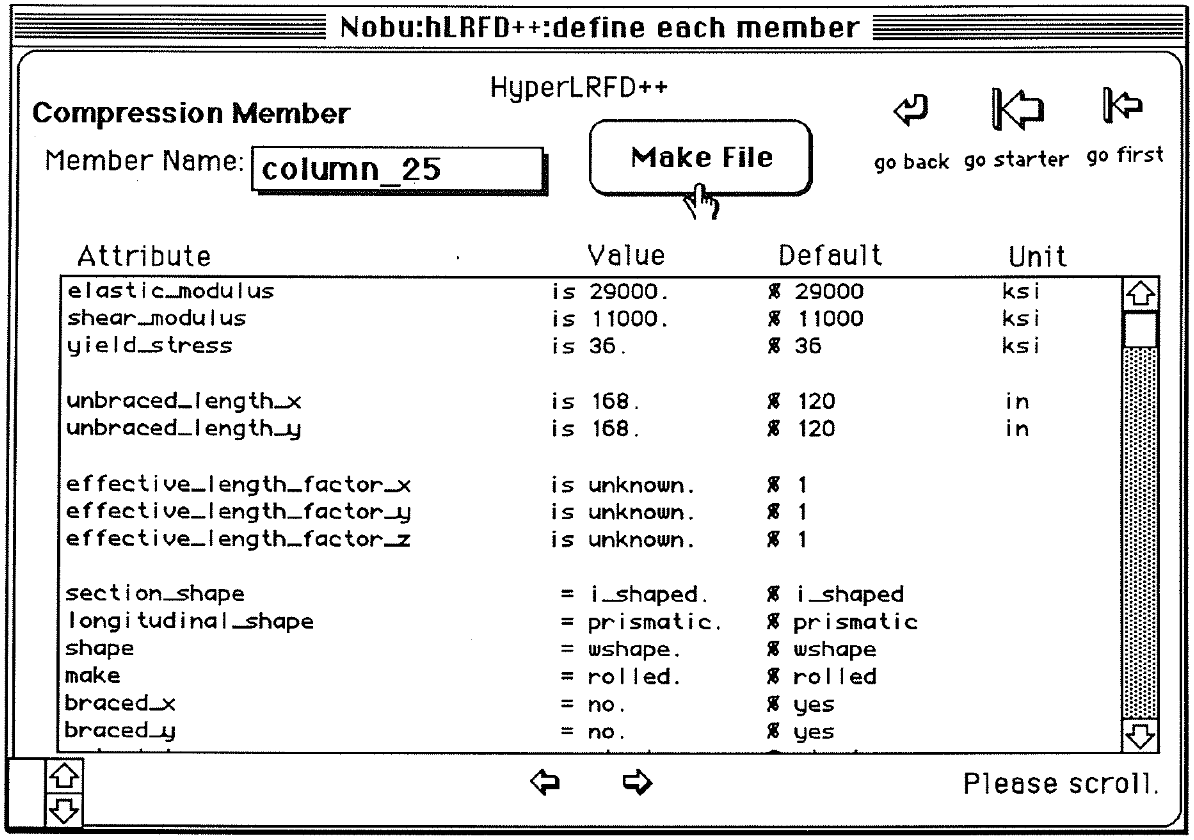


Figure 6-5 The Object Data Model and "Column_25"

Once the member “column_25” is defined as an instance of “steel column,” an appropriate table appears as shown in Figure 6-6 to allow the user to enter the name, attribute values, and external constraints (if any) for the design member. Note that not all the attribute values are required and default values are provided. During the execution of the program, the system will request the user to enter the attribute values that are needed but are not given or determined earlier. Assume that the required strength of the column is given as 720 kips and that the unbraced lengths of both x and y axes are 14 ft = 168 in. Figure 6-7 shows a summary of the attribute values assigned to the member “column_25.” By clicking the button “Make File” in Figure 6-6, an input file that is readable by Prolog++ is created for the member.



Member Name: column_25

Attribute	Value	Default	Unit
elastic_modulus	is 29000.	% 29000	ksi
shear_modulus	is 11000.	% 11000	ksi
yield_stress	is 36.	% 36	ksi
unbraced_length_x	is 168.	% 120	in
unbraced_length_y	is 168.	% 120	in
effective_length_factor_x	is unknown.	% 1	
effective_length_factor_y	is unknown.	% 1	
effective_length_factor_z	is unknown.	% 1	
section_shape	= i_shaped.	% i_shaped	
longitudinal_shape	= prismatic.	% prismatic	
shape	= wshape.	% wshape	
make	= rolled.	% rolled	
braced_x	= no.	% yes	
braced_y	= no.	% yes	

Figure 6-6 Defining Member Attributes for “Column_25”

Attribute	Value	Default Value	Unit
elastic_modulus	is 29000.	% 29000	ksi
shear_modulus	is 11000.	% 11000	ksi
yield_stress	is 36.	% 36	ksi
unbraced_length_x	is 168.	% 120	in
unbraced_length_y	is 168.	% 120	in
effective_length_factor_x	is unknown.	% 1	
effective_length_factor_y	is unknown.	% 1	
effective_length_factor_z	is unknown.	% 1	
section_shape	= i_shaped.	% i_shaped	
longitudinal_shape	= prismatic.	% prismatic	
shape	= wshape.	% wshape	
make	= rolled.	% rolled	
braced_x	= no.	% yes	
braced_y	= no.	% yes	
material	= steel.	% steel	
load_tension	is 0.	% 0	kips
load_comp	is 720.	% 300	kips
load_moment_x	is 0.	% 0	kips
load_moment_y	is 0.	% 0	kips
load_shear_x	is 0.	% 0	kips
load_shear_y	is 0.	% 0	kips
load_moment_nt_x	is 0.	% 0	k-in
load_moment_nt_y	is 0.	% 0	k-in
load_moment_lt_x	is 0.	% 0	k-in
load_moment_lt_y	is 0.	% 0	k-in
m1_x	is 0.	% 0	k-in
m2_x	is 0.	% 0	k-in
m1_y	is 0.	% 0	k-in
m2_y	is 0.	% 0	k-in

Figure 6-7 Attributes and Their Values of "Column_25"

6.2.2 Selection of a Trial Designation for the Column

The process in which the Component Design Module selects a trial W shape designation for "column_25" can be summarized as follows:

- The user starts the design/checking programs.
- The system reads the attributes of the member and links it to the "wshape" Data Object.
- The user specifies the depth of W shape member and its yield stress.

- The module selects a trial designation from the table “Columns W Shapes” by the effective length of the y-axis and the required strength.
- The module checks the required strength about the x-axis by computing the equivalent effective length of the x-axis.

These tasks are described in this section.

The user initiates the design/checking process by clicking the third button “Open Design/Checking Program” of the card of HyperCard shown in Figure 6-4. Then, the user sends a message “design(column_25, ID)” to the object “designer” of the Component Design Module in the form of a query:

```
designer<-design(column_25, ID).
```

ID is a variable representing a W shape designation that will be returned by the module.

The system then reads in the input file for the design module as prepared earlier. Since the “shape” attribute value is “wshape,” the module links the object “column_25” to the Data Object “wshape,” and then opens the database containing W shape dimensions and properties.

The Component Design Module for W shape compression members consists of heuristics for finding the least weight W shape designation of compression members. The heuristics is based on the “columns W shapes” table given in the LRFD Manual [AISC 86]. The table consists of a set of designations, effective lengths of the y-axis, and corresponding design compressive strengths based on the limit state of member buckling about the y-axis. The table can be further divided into multiple tables according to the depth and yield stress F_y of member sections. Table 6-1 shows an example of the table of W14 sections with yield stress F_y = 36 ksi. The table is currently stored in the module in the form of Prolog++ sentences such as:

```
column(w14x90,[(0,810),(6,795),(7,789),(8,783), ... , (38,365)]).
column(w14x99,[(0,890),(6,873),(7,867),(8,860), ... , (38,402)]).
```

Table 6-1 Design Compressive Strength for W14 Columns (F_y = 36 ksi)

KyLy (ft)	....	W14x90	W14x99	W14x109	....
0	....	810	890	979	....
6	....	795	873	960	....
7	....	789	867	950	....
8	....	783	860	946	....
9	....	775	852	937	....
10	....	767	843	927	....
11	....	758	833	917	....
12	....	749	823	905	....
13	....	738	811	893	....
14	....	728	799	880	....
15	....	716	787	866	....
16	....	704	773	852	....
17	....	691	759	837	....
18	....	678	745	821	....
19	....	664	730	804	....
....	....	....	....	....	....

In order to select a trial section, the effective length of y-axis of the column is needed. Since the effective length factor K_y is not given, the module asks the user to enter the value. If necessary, the user can first read the provision about the effective length factor. In this preliminary design example, the user simply gives K_y = 1.2 because the frame is unbraced. Since the values that the user enters during the program execution are stored in the computer memory, the system will not ask the user again during the session. However, they are not stored in the Design Member Object permanently. Thus, when the user checks the design in the future, the system will ask for these values unless the user changes the attribute values of the Design Member Object. The module can now try to find a least weight W14 section that satisfies the condition that the design compressive strength is greater than P_u = 720 kips for effective length, K_yL_y = 16.8 ft. From Table 6-1, W14x99 is chosen as the candidate section. This table can also be stored in a relational database system. A query can be issued to retrieve the candidate section.

Finally, the module checks the design strength of the member about the x-axis by computing the equivalent effective length of x-axis:

$$K_xL_{x\_eq} = K_xL_x \ / \ (r_x \, / \, r_y)$$

Note that if the equivalent effective length $K_xL_{x\_eq}$ is smaller than the effective length about the weak y-axis K_yL_y, the selected member will satisfy the required strength automatically. Otherwise, new selection is needed based on the equivalent effective length $K_xL_{x\_eq}$. Since K_x is unknown, the system asks the user to enter the value. In this example, the user enters the value of K_x as 1.2. The radii of gyration, r_x and r_y, about the x and y axes, are automatically obtained from the table "WSHAPE_DIM" in the Oracle database system. In this example, we obtain,

$$K_xL_{x\_eq} = 8.44 < 16.8 = K_yL_y$$

Since $K_xL_{x\_eq}$ is smaller than K_yL_y, buckling about the weak y-axis governs and the W14x99 remains as a feasible candidate. The system can now proceed to check this trial designation according to appropriate design provisions.

6.2.3 Conformance Checking of a Trial Design

This section presents the checking of the trial design selected in the previous section for compliance with design provisions. The procedure for conformance checking can be summarized as follows:

- The user selects a specific behavior state as a focused class in the Member Class Hierarchy.
- The module checks if the user-selected class is appropriate.
- The module traverses the subtree of the selected class to identify applicable requirements.
- The system executes the identified applicable requirements.

This procedure is described in detail in this section.

In conformance checking, the system distinguishes whether the design stage is a preliminary or detailed design. For detailed design, the system checks all the appropriate design provisions. The user can also perform preliminary design checking by focusing on a class of a specific behavior state. In this example, the user specifies for preliminary design of members. As noted in Figure 6-8, the user selects the class "non_slender_ prism_y_comp_mem" by clicking the items "Class," "comp_mem," and "y_buckle" from the hierarchically structured menu.

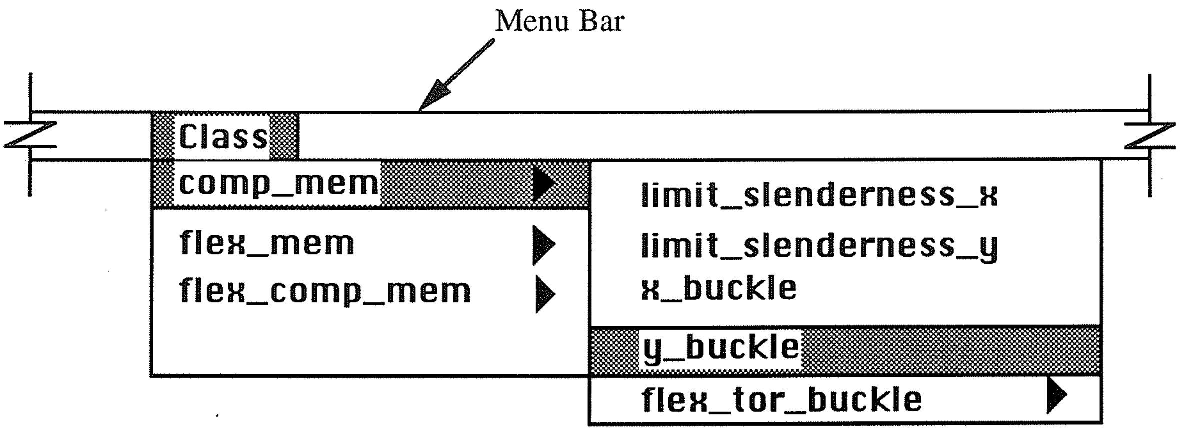


Figure 6-8 The Hierarchical Menu for Selecting a Focused Class in Preliminary Design

As described in Section 6.1.1, the design checking module identifies whether the selected class is appropriate for this member by traversing backward from the selected class to the root node class "member" of the Member Class Hierarchy. Once the selected class is affirmed, its subclasses are traversed to determine the appropriate requirements. This classification process is shown in Figure 6-9. In this example, the method "clas_non_slender_prism_y_comp_mem," which is linked to the class "non_slender_prism_y_comp_mem," will find that the member "column_25" is a "short_non_slender_prism_y_comp_mem" since the column slenderness parameter λ_c is less than 1.5. Furthermore, the member "column_25" is linked to the leaf class "short_non_slender_prism_y_comp_mem," which is the requirement to be checked for the member.

6.2.4 Post-Processing

After the Component Design Module determines the designation, the system shows the result to the user in a tabulated form as shown in Figure 6-10. The provisions that have been checked are displayed in the table. In this example, the member with W14x99 is checked to be satisfied for the requirement. Figure 6-11 summarizes the steps of the session for designing the column. In this figure the user input is indicated by bold face. Finally, the variable ID in the query "designer<-design(column_25,ID)" is returned with a

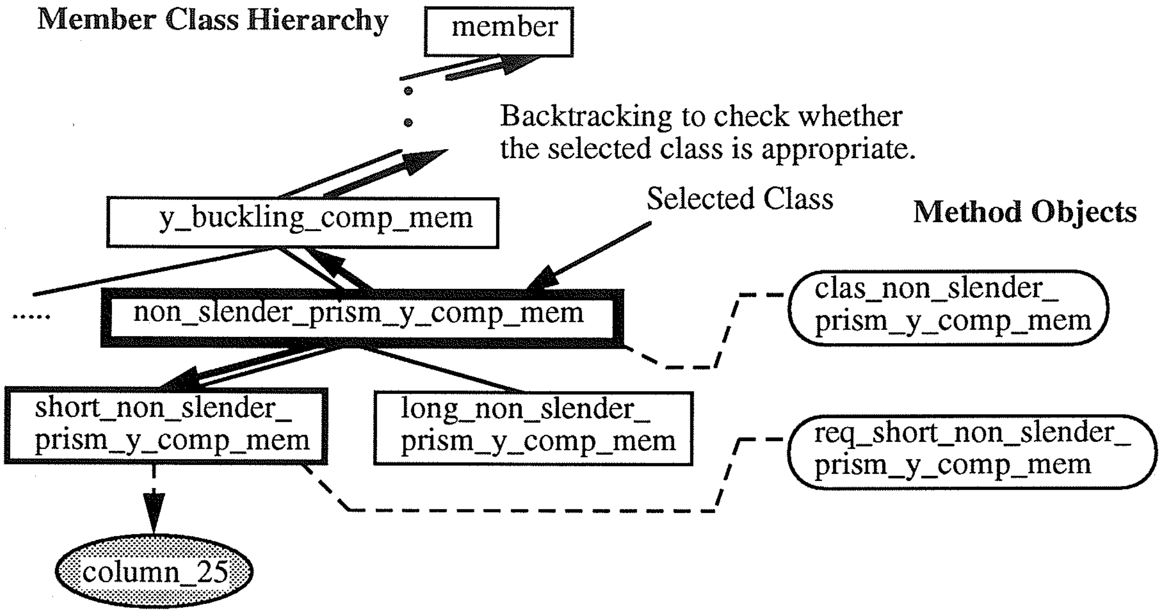
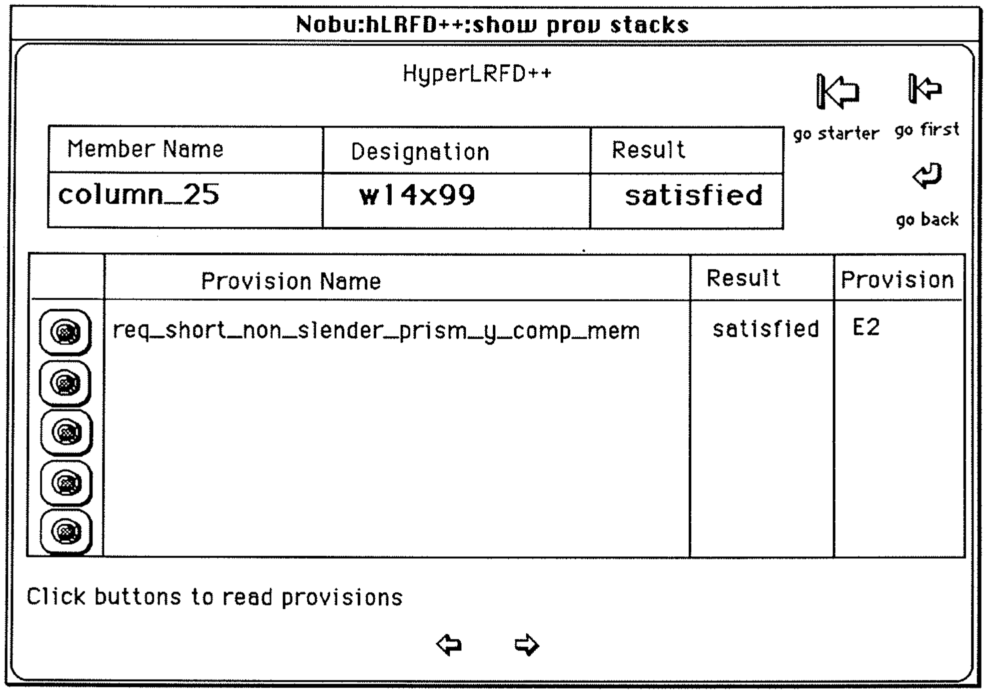


Figure 6-9 Member Class Hierarchy Traversal in Preliminary Design



Nobu:hLRFD++:show prov stacks

HyperLRFD++

Member Name	Designation	Result
column_25	w14x99	satisfied

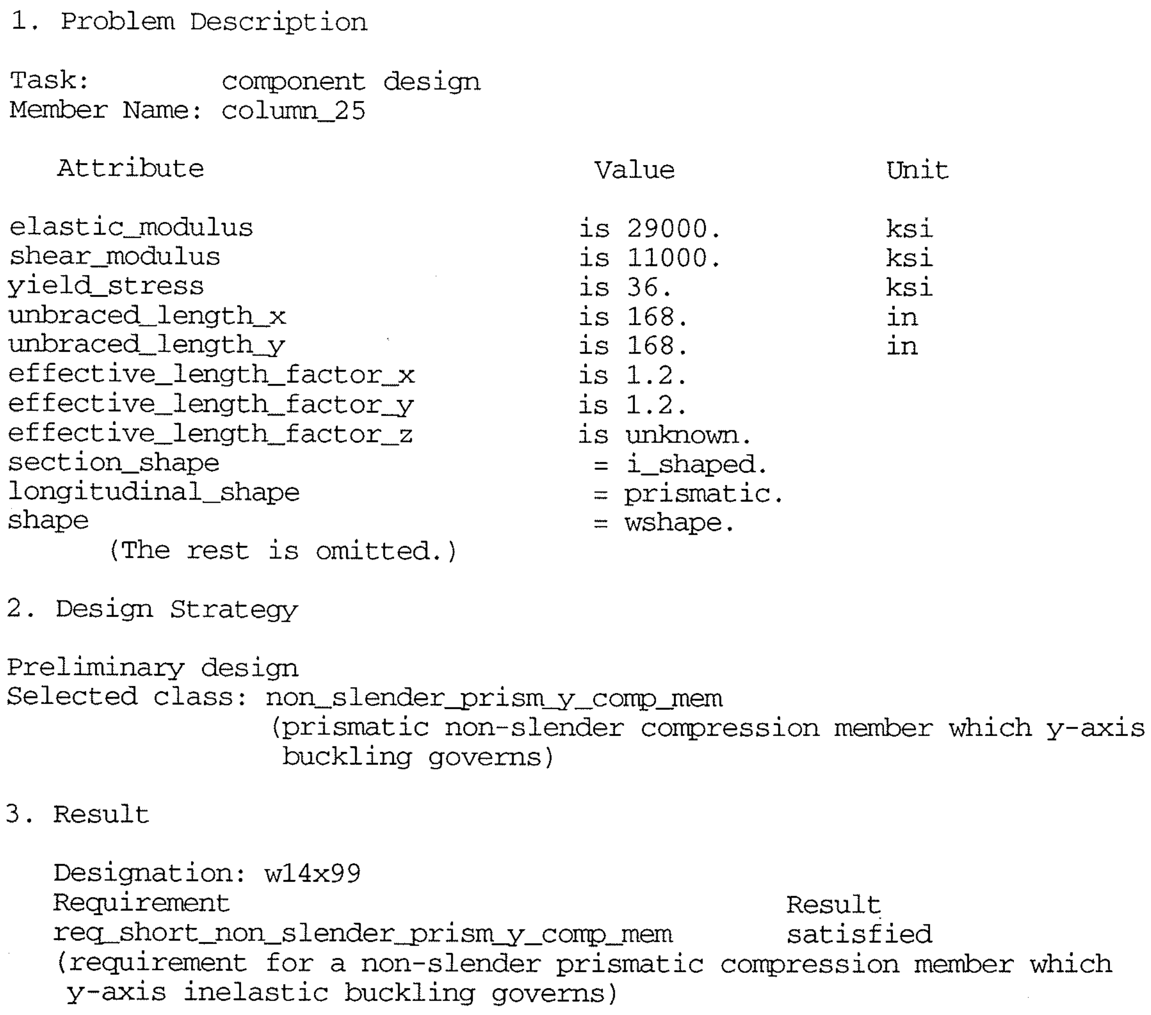
	Provision Name	Result	Provision
	req_short_non_slender_prism_y_comp_mem	satisfied	E2

Click buttons to read provisions

Figure 6-10 The Table Showing the Result of Designing Column_25

```
:-  designer<-design(column_25,  ID)

*****  TRIAL SECTION SELECTION  *****
Designing wshape_comp_mem
Enter a designation (default: w14: enter d) : d
Please input effective_length_factor_y.
Do you want to read the corresponding provision? (y/n):n
Enter effective_length_factor_y (default is 1 :enter d) = 1.2
Please input effective_length_factor_x.
Do you want to read the corresponding provision? (y/n):n
Enter effective_length_factor_x (default is 1 :enter d) = 1.2
KyLy = 16.8
First Trial Section ID = w14x99
KxLx_eq = 8.44
OK
TRIAL SECTION ID = w14x99

*****  CHECKING THE TRIAL SECTION  *****
Preliminary Design or Detailed Design? (p/d) = p
Please enter the class representing the member
Class name = non_slender_prism_y_comp_mem
***  Checking the selected class by traversing the hierarchy backward:
classified as non_slender_prism_y_comp_mem
classified as non_slender_prism_comp_mem
classified as prism_comp_mem
classified as comp_mem
classified as steel_mem
The selected class is OK
***  Traversing the hierarchy from the selected class:
classified as short_non_slender_prism_y_comp_mem
Requirement List = [req_short_non_slender_prism_y_comp_mem]

***  Requirement req_short_non_slender_prism_y_comp_mem is being
evaluated.
req_short_non_slender_prism_y_comp_mem is SATISFIED.

Do you want to read the provisions? (y/n) : y
Please change to HyperCard by clicking the Apple menu.
Then, click the #4 button of "Show Provisions."

***  RESULT  ***
*****  w14x99  is the design  *****
Nº1         ID = w14x99
```

Figure 6-11 The Session of Designing "Column_25"



```
1. Problem Description

Task:          component design
Member Name:   column_25

   Attribute                             Value          Unit

elastic_modulus                        is 29000.        ksi
shear_modulus                          is 11000.        ksi
yield_stress                           is 36.           ksi
unbraced_length_x                      is 168.          in
unbraced_length_y                      is 168.          in
effective_length_factor_x              is 1.2.
effective_length_factor_y              is 1.2.
effective_length_factor_z              is unknown.
section_shape                           = i_shaped.
longitudinal_shape                      = prismatic.
shape                                   = wshape.
      (The rest is omitted.)

2. Design Strategy

Preliminary design
Selected class: non_slender_prism_y_comp_mem
                (prismatic non-slender compression member which y-axis
                 buckling governs)

3. Result

   Designation: w14x99
   Requirement                                   Result
   req_short_non_slender_prism_y_comp_mem        satisfied
   (requirement for a non-slender prismatic compression member which
    y-axis inelastic buckling governs)
```

Figure 6-12 Design Report for the Member "Column_25"

value w14x99. Based on the user's request, the system generates a design report, which consists of:

- problem description, which contains the name, attribute values, and external constraints of the design member, as shown in Figure 6-7,
- design strategy, which shows either the preliminary or detailed design and assumptions given by the user,
- result, which shows checked requirements and their results of either satisfied or violated.

A part of the design report for the member "column_25" is shown in Figure 6-12.

6.3 Conformance Checking for Detailed Design of a Column

This section describes a second example for checking all the appropriate provisions that are applicable to the column designed in the previous section. This example illustrates how external programs and documents in the HyperDocument system may help the user in the design process. As shown in Figure 6-13, the required compressive strength of the column is larger than the one in the preliminary design (because of the change in load condition.) Furthermore, the beams are provided to be W21x50, rigidly connected to the column in both direction.

The design checking procedure is divided into three basic steps:

1. Problem definition: the user defines all known attributes of the example problem.
2. Conformance checking: the system determines all appropriate classes and requirements, given the description of the column.
3. Re-design: the user re-designs the column by exploring provisions and background information.

These tasks are described in the following subsections.

6.3.1 Problem Definition

Defining this example problem involves two basic steps:

- modifying the data file for the column, and
- accessing the attribute values of the column from the database.

First, the user opens the file representing column_25, then changes the "load_comp" attribute value from 720 to 770 in the member definer table shown in Figure 6-6.

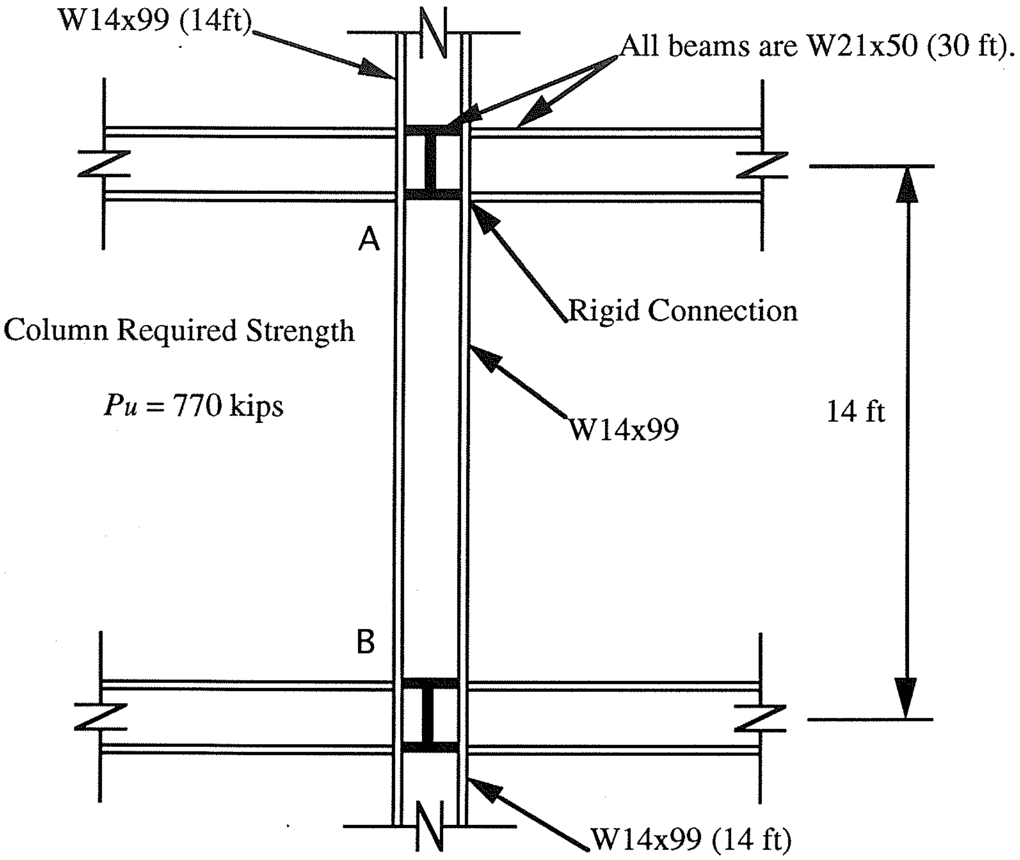


Figure 6-13 The Column in the Detailed Design Phase

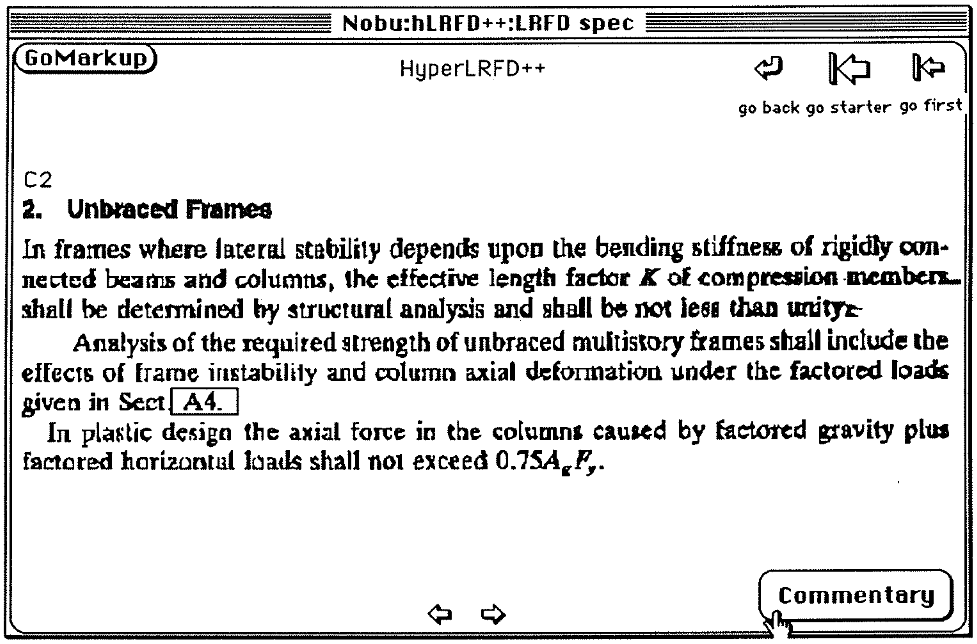
6.3.2 Conformance Checking

This section presents the conformance checking procedure. After opening the design/checking programs, the user issues a query:

```
checker<-check(column_25, w14x99, Result)
```

where `Result` is a variable that "satisfied" or "violated" will be instantiated. Then, the Conformance Checking Module is executed. The module links "column_25" to the Data Object "wshape," then asks the user if it is a preliminary or detailed design phase.

In the detailed design process, the system traverses the Member Class Hierarchy from the root node "member." During this traversal and classification process, the effective length factor K_x is unknown and needed. The system asks the user to enter the value. In this example, the user requests the system to show the provision explaining the effective length factor. By the pointer from the Method Object "k_factor_x" to the provision document "C2," the system shows the provision to the user as in Figure 6-14. The user can further read its commentary "C-C2" by clicking the "Commentary" button on the provision document. The commentary on this provision is quite long and is spread over several cards. As the user browses through these cards, the alignment chart for determining the effective length factor K of unbraced frame appears on the screen as shown in Figure 6-15. As noted in the lower corner of Figure 6-15, there is an external program that can be used to calculate the K-factor. The user can click the button "K-factor" to open the Excel spreadsheet program for determining K as shown in Figure 6-16.



C2

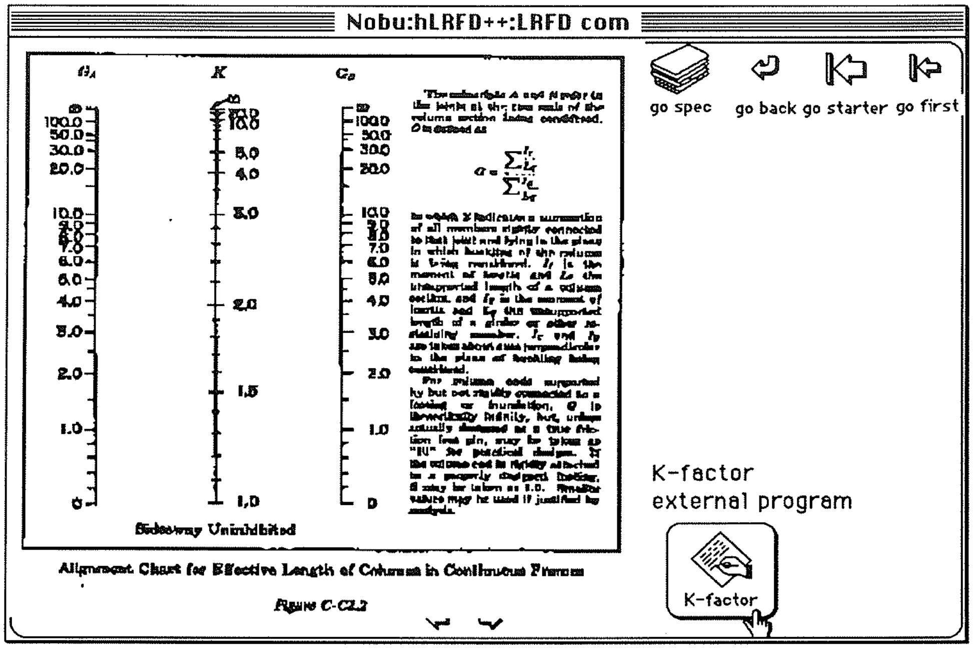
2. Unbraced Frames

In frames where lateral stability depends upon the bending stiffness of rigidly connected beams and columns, the effective length factor K of compression members shall be determined by structural analysis and shall be not less than unity.

Analysis of the required strength of unbraced multistory frames shall include the effects of frame instability and column axial deformation under the factored loads given in Sect A4.

In plastic design the axial force in the columns caused by factored gravity plus factored horizontal loads shall not exceed $0.75A_gF_y$.

Figure 6-14 Provision Document "C2"



The subscripts A and B refer to the joints at the two ends of the column section being considered. G is defined as

$$G = \frac{\sum \frac{I_c}{L_c}}{\sum \frac{I_g}{L_g}}$$

in which Σ indicates a summation of all members rigidly connected to that joint and lying in the plane in which buckling of the column is being considered. I_c is the moment of inertia and L_c the unsupported length of a column section, and I_g is the moment of inertia and L_g the unsupported length of a girder or other restraining member. I_c and I_g are taken about axes perpendicular to the plane of buckling being considered.

For column ends supported by but not rigidly connected to a footing or foundation, G is theoretically infinity, but, unless actually designed as a true friction free pin, may be taken as "10" for practical design. If the column end is rigidly attached to a properly designed footing, G may be taken as 1.0. Smaller values may be used if justified by analysis.

Alignment Chart for Effective Length of Columns in Continuous Frames

Figure C-C2.2

Figure 6-15 The Document Showing the Alignment Chart with a Button

As shown in Figure 6-16, based on the input data (designations, lengths, and considering axes of the columns and beams) provided by the user, the spreadsheet program automatically computes the restraint factors G_A and G_B for the end joints of the column and the effective length factor K. The K_x value is sent to the Method Object "k_factor_x" after the spreadsheet program is closed. The effective length factor K_y about the weak y-axis can be calculated in a similar manner. (In this example, the K_x and K_y values are determined to be 1.6 and 1.2 respectively.)

After the traversal of the Member Class Hierarchy is complete, the column is classified as the following types:

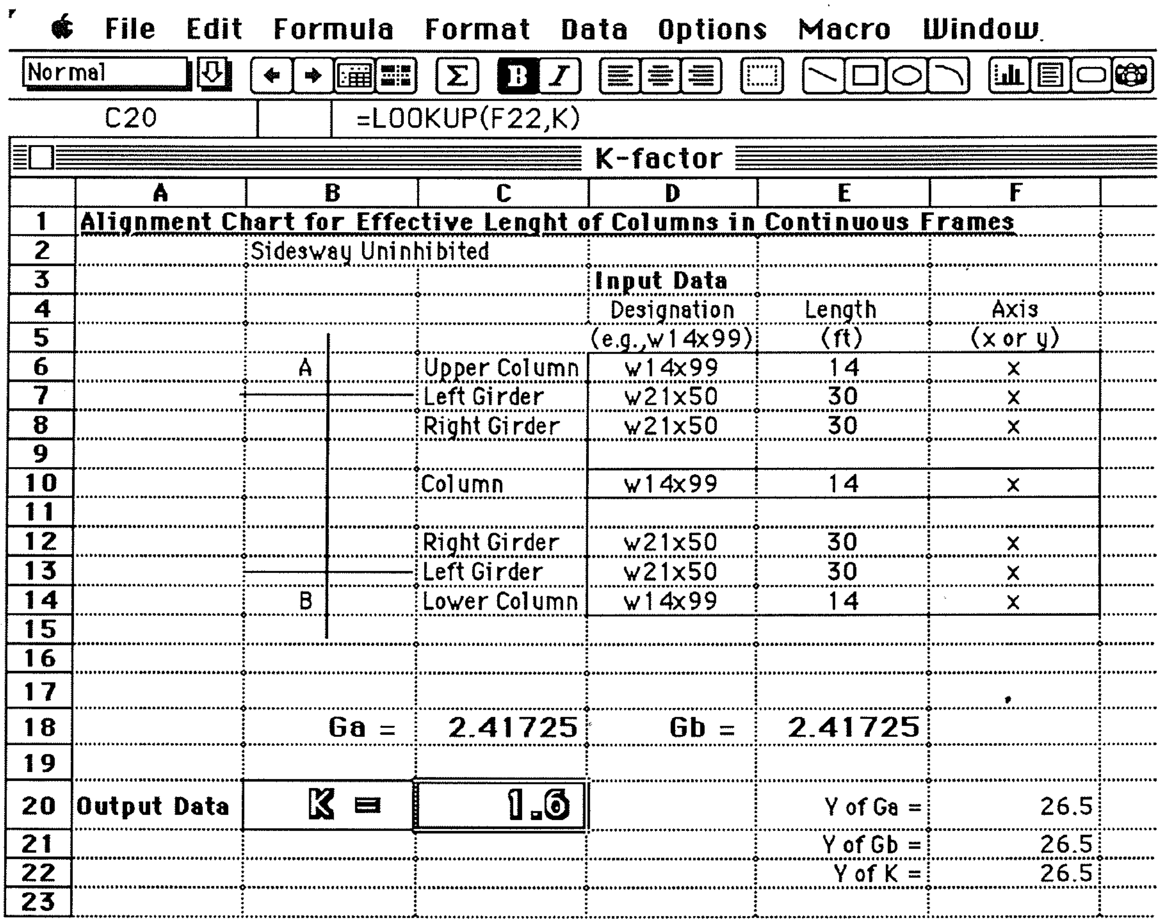
- short_non_slender_prism_y_comp_mem,
- short_non_slender_ftb_prism_comp_mem,
- limit_slender_x_comp_mem, and
- limit_slender_y_comp_mem,

Furthermore, the applicable requirements of these leaf node classes are recorded:

- req_short_non_slender_prism_y_comp_mem,

- req_short_non_slender_ftb_prism_comp_mem,
- req_limit_slenderness_ratio_x_comp_mem, and
- req_limit_slenderness_ratio_y_comp_mem.

The system executes the Method Objects of each requirement for conformance checking. The results are tabulated as shown in Figure 6-17. Figure 6-18 lists the details of the session on the conformance checking of the column. A part of the design report is shown in Figure 6-19. As the result indicates the requirement "req_short_non_slender_prism_y_comp_mem" is violated. The next section describes the re-design process that takes advantages of the availability of the HyperDocument system.

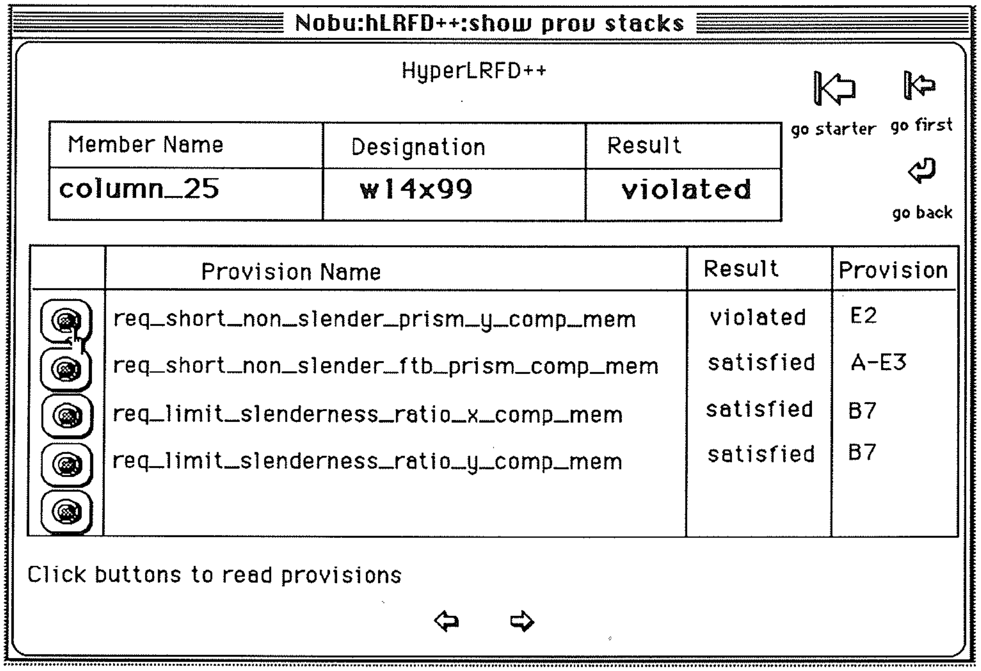


C20 =LOOKUP(F22,K)

K-factor

	A	B	C	D	E	F
1	Alignment Chart for Effective Lenght of Columns in Continuous Frames					
2		Sidesway Uninhibited				
3				Input Data		
4				Designation	Length	Axis
5				(e.g.,w14x99)	(ft)	(x or y)
6		A	Upper Column	w14x99	14	x
7			Left Girder	w21x50	30	x
8			Right Girder	w21x50	30	x
9						
10			Column	w14x99	14	x
11						
12			Right Girder	w21x50	30	x
13			Left Girder	w21x50	30	x
14		B	Lower Column	w14x99	14	x
15						
16						
17						
18		Ga =	2.41725	Gb =	2.41725	
19						
20	Output Data	K =	1.6		Y of Ga =	26.5
21					Y of Gb =	26.5
22					Y of K =	26.5
23						

Figure 6-16 The Screen Image of the K-factor Program of Excel



Member Name	Designation	Result
column_25	w14x99	violated

	Provision Name	Result	Provision
	req_short_non_slender_prism_y_comp_mem	violated	E2
	req_short_non_slender_ftb_prism_comp_mem	satisfied	A-E3
	req_limit_slenderness_ratio_x_comp_mem	satisfied	B7
	req_limit_slenderness_ratio_y_comp_mem	satisfied	B7

Click buttons to read provisions

Figure 6-17 The Conformance Checking Result of "Column_25"

6.3.3 Re-design of the Column

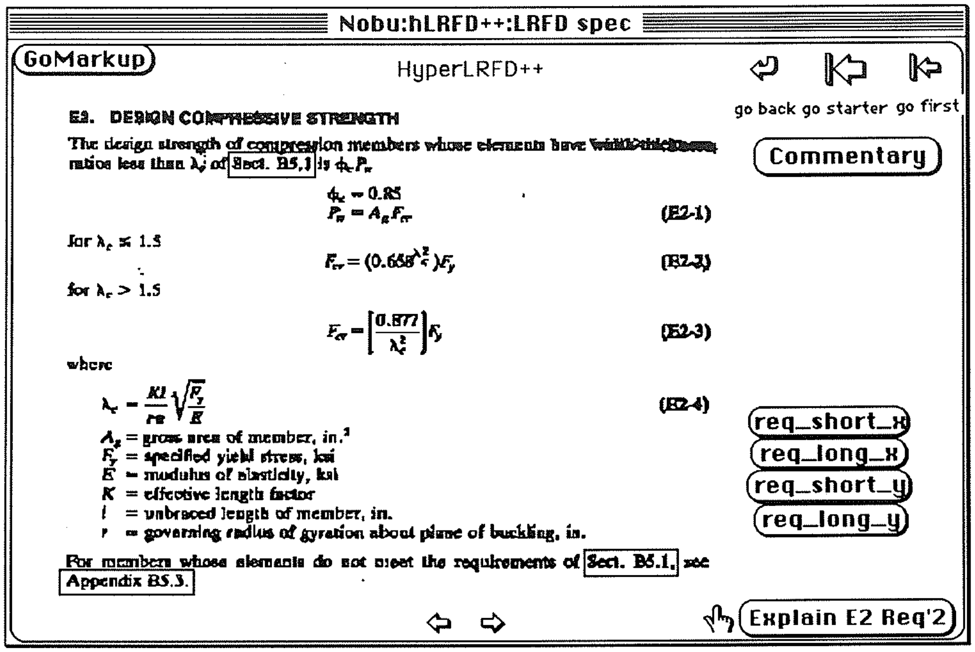
This section describes the use of the HyperDocument system for re-design. Continuing the example discussed in the previous section, the W14x99 column violates the requirement "req_short_non_slender_prism_y_comp_mem" of Provision E2. To understand the nature of the requirement, the user can proceed to browse the provision E2 shown in Figure 6-20, by clicking the button next to the requirement name in Figure 6-17. By clicking the button "Explain E2 Req'2" on the provision the user can obtain the specific meaning for the requirements as shown in Figure 6-21. Now, the user realizes that the governing limit state of the column is inelastic buckling through the explanation.

```
:-  checker<-check(column_25, w14x99, Result)

Preliminary Design or Detail Design? (p/d) = d
***  Traversing the hierarchy:
classified as steel_mem
classified as comp_mem
classified as prism_comp_mem
classified as non_slender_prism_comp_mem
Please input effective_length_factor_x.
Do you want to read the corresponding provision? (y/n):y
Did you use the External Program K-factor? (y/n):y
K_factor_x = 1.6 is returned.
Please input effective_length_factor_y.
Do you want to read the corresponding provision? (y/n):y
Did you use the External Program K-factor? (y/n):y
K_factor_y = 1.2 is returned.
classified as non_slender_prism_y_comp_mem
classified as short_non_slender_prism_y_comp_mem
classified as non_slender_ftb_prism_comp_mem
Enter effective_length_factor_z (default is 1 :enter d) = d
1
classified as short_non_slender_ftb_prism_comp_mem
classified as limit_slender_x_comp_mem
classified as limit_slender_y_comp_mem

Requirement List = [req_short_non_slender_prism_y_comp_mem,
                    req_short_non_slender_ftb_prism_comp_mem,
                    req_limit_slenderness_ratio_x_comp_mem,
                    req_limit_slenderness_ratio_y_comp_mem]

***   Requirement  req_short_non_slender_prism_y_comp_mem  is  being
evaluated.
req_short_non_slender_prism_y_comp_mem is VIOLATED.

***  Requirement req_short_non_slender_ftb_prism_comp_mem is being
evaluated.
req_short_non_slender_ftb_prism_comp_mem is SATISFIED.

***   Requirement  req_limit_slenderness_ratio_x_comp_mem  is  being
evaluated.
req_limit_slenderness_ratio_x_comp_mem is SATISFIED.

***   Requirement  req_limit_slenderness_ratio_y_comp_mem  is  being
evaluated.
req_limit_slenderness_ratio_y_comp_mem is SATISFIED.

***  Final Result is VIOLATED  *****
Do you want to read the provisions? (y/n) : y
Please change to HyperCard by clicking the Apple menu.
Then, click the #4 button of "Show Provisions."
Nº1         Result = violated
```

Figure 6-18 The Session of Conformance Checking of "Column_25"

```
1. Problem Description

Task:         conformance checking
Member Name: column_25
Designation: w14x99

   Attribute                          Value            Unit

elastic_modulus                      is 29000.         ksi
shear_modulus                        is 11000.         ksi
yield_stress                         is 36.            ksi
unbraced_length_x                    is 168.           in
unbraced_length_y                    is 168.           in
effective_length_factor_x            is 1.6.
effective_length_factor_y            is 1.2.
effective_length_factor_z            is unknown.
section_shape                         = i_shaped.
longitudinal_shape                    = prismatic.
shape                                 = wshape.
      (The rest is omitted.)

2. Design Strategy

Detailed design

3. Result

   Requirement                                    Result
   req_short_non_slender_prism_y_comp_mem         violated
   (requirement for a non-slender prismatic compression member which
    y-axis inelastic buckling governs)
   req_torsional_buckling_comp_mem                satisfied
   (requirement for a compression member whose limit state is torsional
    buckling)
   req_limit_slenderness_ratio_x_comp_mem         satisfied
   (requirement for a limit slenderness of the x-axis for a compression
    member)
   req_limit_slenderness_ratio_y_comp_mem         satisfied
   (requirement for a limit slenderness of the y-axis for a compression
    member)
```

Figure 6-19 Design Report for the Member "Column_25"



E2. DESIGN COMPRESSIVE STRENGTH

The design strength of compression members whose elements have width-thickness ratios less than λ_r of Sect. B5.1 is $\phi_c P_n$

$$\phi_c = 0.85$$

$$P_n = A_g F_{cr} \qquad \text{(E2-1)}$$

for $\lambda_c \leq 1.5$

$$F_{cr} = (0.658^{\lambda_c^2}) F_y \qquad \text{(E2-2)}$$

for $\lambda_c > 1.5$

$$F_{cr} = \left[\frac{0.877}{\lambda_c^2}\right] F_y \qquad \text{(E2-3)}$$

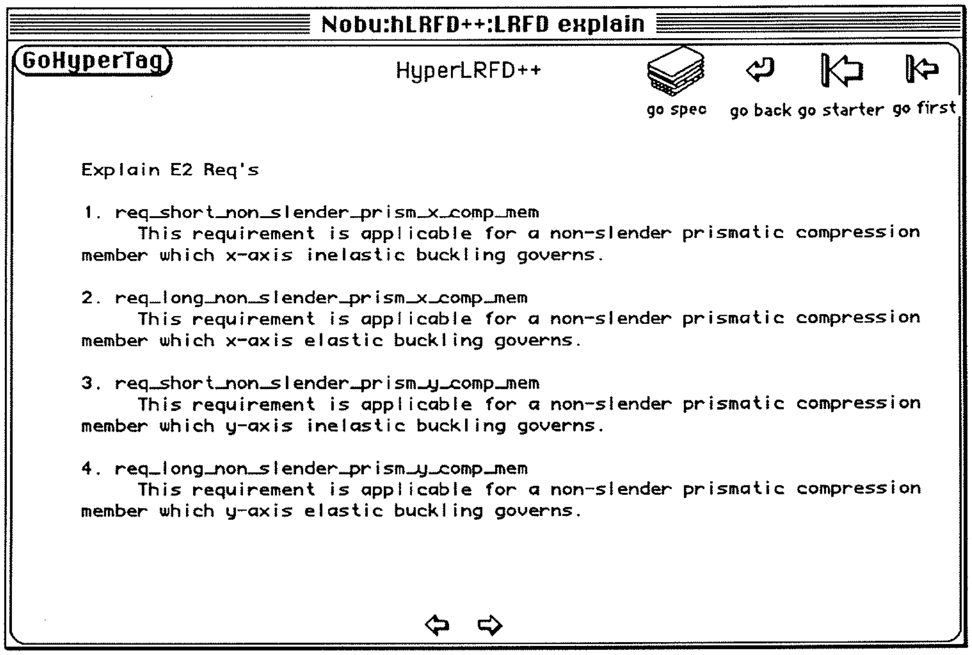
where

$$\lambda_c = \frac{Kl}{r\pi}\sqrt{\frac{F_y}{E}} \qquad \text{(E2-4)}$$

A_g = gross area of member, in.^{2}
F_y = specified yield stress, ksi
E = modulus of elasticity, ksi
K = effective length factor
l = unbraced length of member, in.
r = governing radius of gyration about plane of buckling, in.

For members whose elements do not meet the requirements of Sect. B5.1, see Appendix B5.3.

Figure 6-20 Provision E2 from the AISC LRFD specification [AISC 86]



Explain E2 Req's

1. req_short_non_slender_prism_x_comp_mem
This requirement is applicable for a non-slender prismatic compression member which x-axis inelastic buckling governs.

2. req_long_non_slender_prism_x_comp_mem
This requirement is applicable for a non-slender prismatic compression member which x-axis elastic buckling governs.

3. req_short_non_slender_prism_y_comp_mem
This requirement is applicable for a non-slender prismatic compression member which y-axis inelastic buckling governs.

4. req_long_non_slender_prism_y_comp_mem
This requirement is applicable for a non-slender prismatic compression member which y-axis elastic buckling governs.

Figure 6-21 Explanation for the Requirements of Provision E2 [AISC 86]

The user can go back to the provision E2 by clicking the button "go back" as shown in Figure 6-21, and then proceed to the commentary C2 by clicking the button "commentary" from the provision E2 shown in Figure 6-20. As shown in Figure 6-22, the commentary C2 describes that the alignment chart is based on the assumption that behavior is purely elastic. As noted in the commentary,

> "Where the actual conditions differ from these assumptions, unrealistic designs may result. There are design procedures available [52, 53] which may be used in the calculation of G for use in Fig. C-C2.2 to give results more truly representative of conditions in real structures." [AISC 86]

This fact suggests that using the alignment chart alone may be inappropriate. The commentary C2 also refers to two research papers. The user can browse through segments of these research papers by clicking the buttons on the reference numbers. For example, Figure 6-23 shows a portion of the research article written by Yura [Yura 71] for the reference 52.

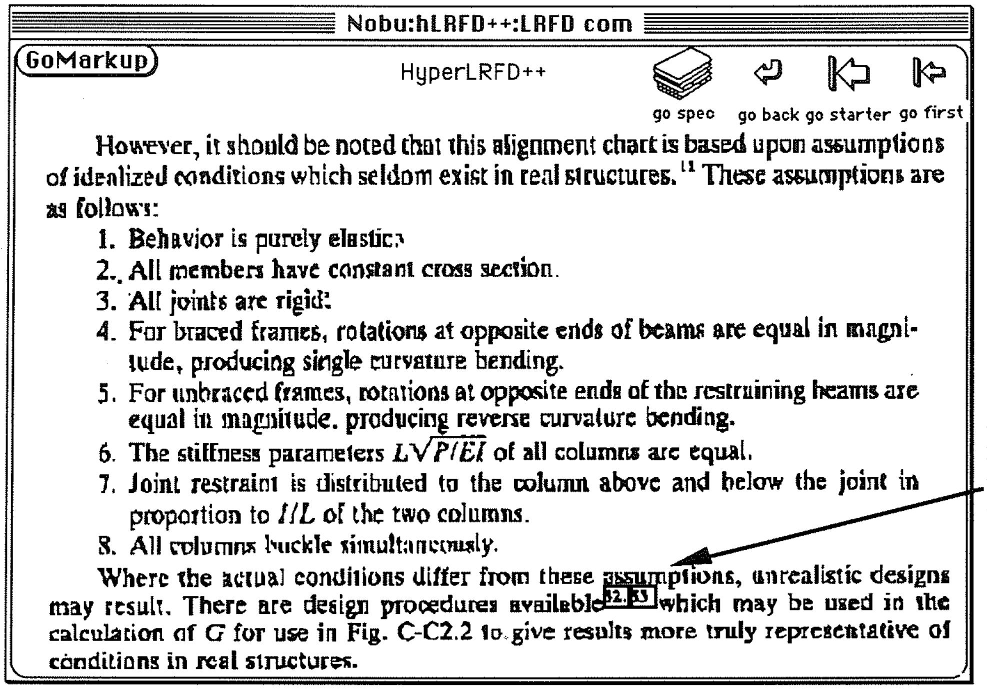
As noted in this article, stiffness reduction factors may be used to calculate the restraint factors G_A and G_B. The user can issue the keyword "stiffness reduction factor" in the HyperDocument system and obtain a document on "Stiffness Reduction Factors" of the LRFD Manual [AISC 86], as shown in Figure 6-24. An external spreadsheet program for calculating stiffness reduction factors is shown in Figure 6-25. By entering the required compressive strength P_u (= 770 kips) and the designation of the column (W14x99), the "SRF" program computes the stiffness reduction factor as $\beta_s = 0.614$. Thus, the restraint factor is obtained as

$$G_{inelastic} = G_{elastic}\ \beta_s = 2.42 * 0.614 = 1.49$$

Then, the user can now return to the spreadsheet program for calculating the K-factor and enter $G = 1.49$ into the cells of G_A and G_B. The effective length factor is thus obtained as:

$$K_x = 1.3$$

In the same manner, the K-factor for the y-axis K_y is determined to be 1.1. The user can re-execute conformance checking of the column, using these effective length factors. In this example, the W14x99 section is found to satisfy all the applicable requirements. This example demonstrates how the HyperDocument system can be used when the user performs re-design due to the requirement violations.



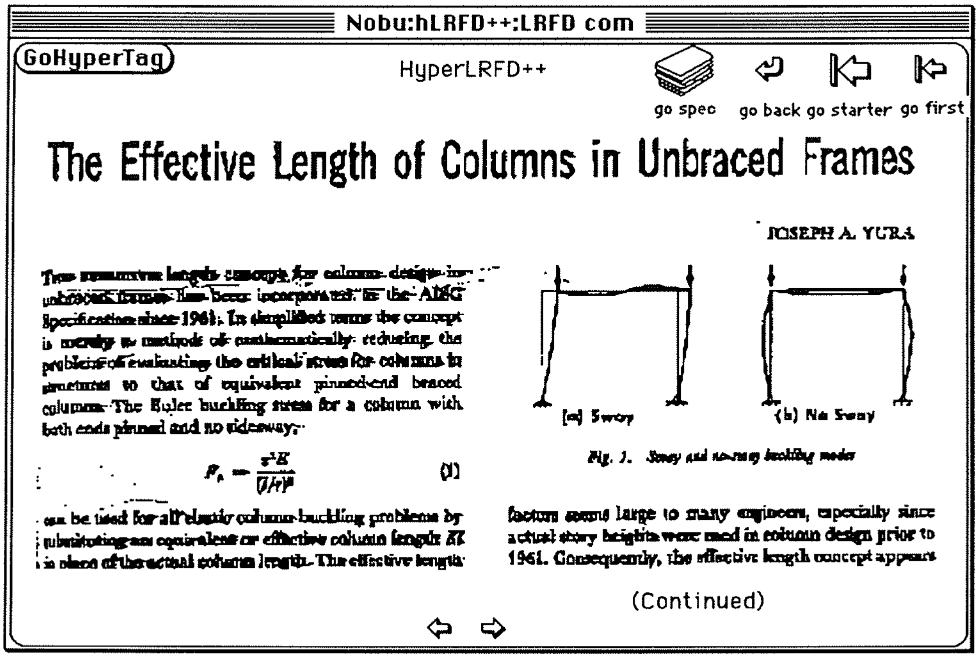
However, it should be noted that this alignment chart is based upon assumptions of idealized conditions which seldom exist in real structures.[11] These assumptions are as follows:

1. Behavior is purely elastic.
2. All members have constant cross section.
3. All joints are rigid.
4. For braced frames, rotations at opposite ends of beams are equal in magnitude, producing single curvature bending.
5. For unbraced frames, rotations at opposite ends of the restraining beams are equal in magnitude, producing reverse curvature bending.
6. The stiffness parameters $L\sqrt{P/EI}$ of all columns are equal.
7. Joint restraint is distributed to the column above and below the joint in proportion to I/L of the two columns.
8. All columns buckle simultaneously.

Where the actual conditions differ from these assumptions, unrealistic designs may result. There are design procedures available [52,53] which may be used in the calculation of G for use in Fig. C-C2.2 to give results more truly representative of conditions in real structures.

Buttons Pointing References 52 and 53

Figure 6-22 A Part of the Commentary C2 of [AISC 86]



The Effective Length of Columns in Unbraced Frames

JOSEPH A. YURA

The effective length concept for column design in unbraced frames has been incorporated in the AISC Specification since 1961. In simplified terms the concept is merely a method of mathematically reducing the problem of evaluating the critical stress for columns in structures to that of equivalent pinned-end braced columns. The Euler buckling stress for a column with both ends pinned and no sidesway,

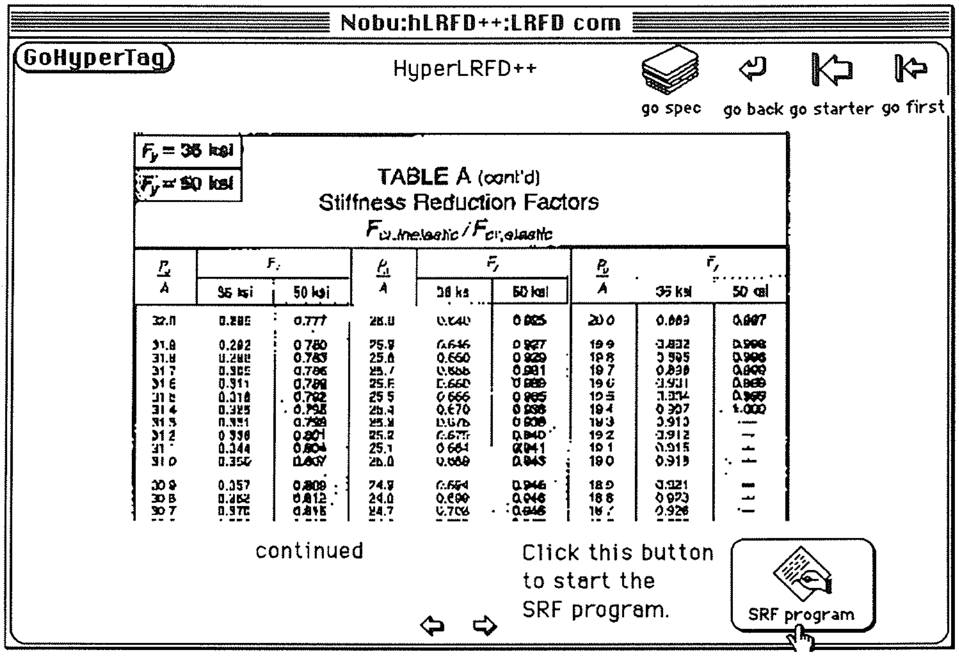
$$F_e = \frac{\pi^2 E}{(l/r)^2} \qquad (1)$$

can be used for all elastic column buckling problems by substituting an equivalent or effective column length Kl in place of the actual column length. The effective length factors seem large to many engineers, especially since actual story heights were used in column design prior to 1961. Consequently, the effective length concept appears

Fig. 1. Sway and nonsway buckling modes

(Continued)

Figure 6-23 A Part of the Paper [Yura 71] Referenced by the Commentary



F_y = 36 ksi

F_y = 50 ksi

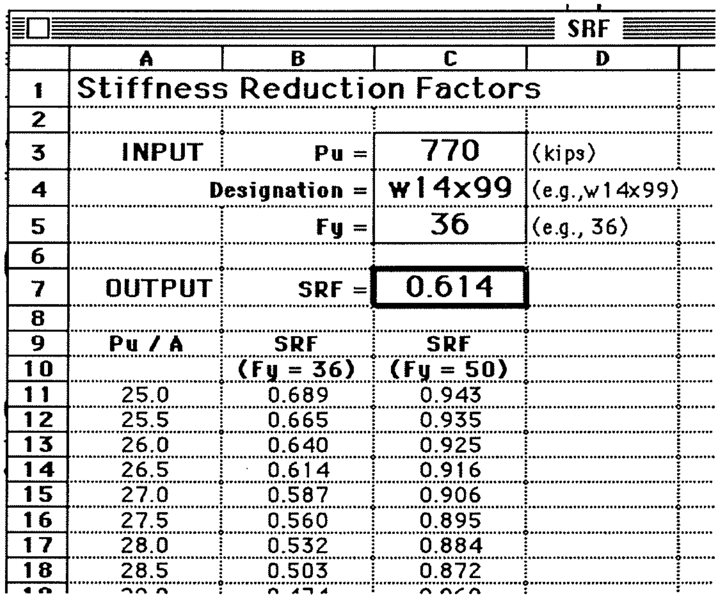
TABLE A (cont'd)
Stiffness Reduction Factors
$F_{cr,inelastic}/F_{cr,elastic}$

$\frac{P_u}{A}$	F_y		$\frac{P_u}{A}$	F_y		$\frac{P_u}{A}$	F_y	
	36 ksi	50 ksi		36 ksi	50 ksi		36 ksi	50 ksi
32.0	0.285	0.777	26.0	0.640	0.925	20.0	0.889	0.997
31.9	0.292	0.780	25.9	0.646	0.927	19.9	0.892	0.998
31.8	0.299	0.783	25.8	0.650	0.929	19.8	0.895	0.998
31.7	0.305	0.786	25.7	0.655	0.931	19.7	0.898	0.999
31.6	0.311	0.789	25.6	0.660	0.933	19.6	0.901	0.999
31.5	0.318	0.792	25.5	0.666	0.935	19.5	0.904	0.999
31.4	0.325	0.795	25.4	0.670	0.936	19.4	0.907	1.000
31.3	0.331	0.798	25.3	0.675	0.938	19.3	0.910	—
31.2	0.338	0.801	25.2	0.679	0.940	19.2	0.912	—
31.1	0.344	0.804	25.1	0.684	0.941	19.1	0.915	—
31.0	0.350	0.807	25.0	0.689	0.943	19.0	0.918	—
30.9	0.357	0.809	24.9	0.694	0.944	18.9	0.921	—
30.8	0.363	0.812	24.8	0.699	0.946	18.8	0.923	—
30.7	0.370	0.815	24.7	0.703	0.948	18.7	0.926	—

continued

Click this button to start the SRF program.

Figure 6-24 HyperDocument Showing a Part of Table A "Stiffness Reduction Factors" [AISC 86]



	A	B	C	D
1	Stiffness Reduction Factors			
2				
3	INPUT	Pu =	770	(kips)
4		Designation =	w14x99	(e.g.,w14x99)
5		Fy =	36	(e.g., 36)
6				
7	OUTPUT	SRF =	0.614	
8				
9	Pu / A	SRF	SRF	
10		(Fy = 36)	(Fy = 50)	
11	25.0	0.689	0.943	
12	25.5	0.665	0.935	
13	26.0	0.640	0.925	
14	26.5	0.614	0.916	
15	27.0	0.587	0.906	
16	27.5	0.560	0.895	
17	28.0	0.532	0.884	
18	28.5	0.503	0.872	

Figure 6-25 A Part of the "SRF" Excel Program

6.4 Conformance Checking of a Flexural Member in the Detailed Design Phase

In this section, an example of checking the beam design with an external constraint is described. Figure 6-26 shows the beam and its design condition. Figure 6-27 summarizes the attribute values of the member "beam_234" defined as an instance of the class "steel beam" in the Object Model. Note that the user gives an external constraint of flange width (bf) less than 10 inches.

After traversing the Member Class Hierarchy, the Conformance Checking Module links the member "beam_234" to the following leaf node classes:

- medium_compact_icb_flex_mem, and
- thick_web_flex_mem.

The requirements corresponding to these classes are:

- req_medium_compact_icb_flex_mem, and
- req_thick_web_flex_mem

The beam is checked for conformance with these requirements and the external constraint. The result is displayed in Figure 6-28, and the design session is given in Figure 6-29.

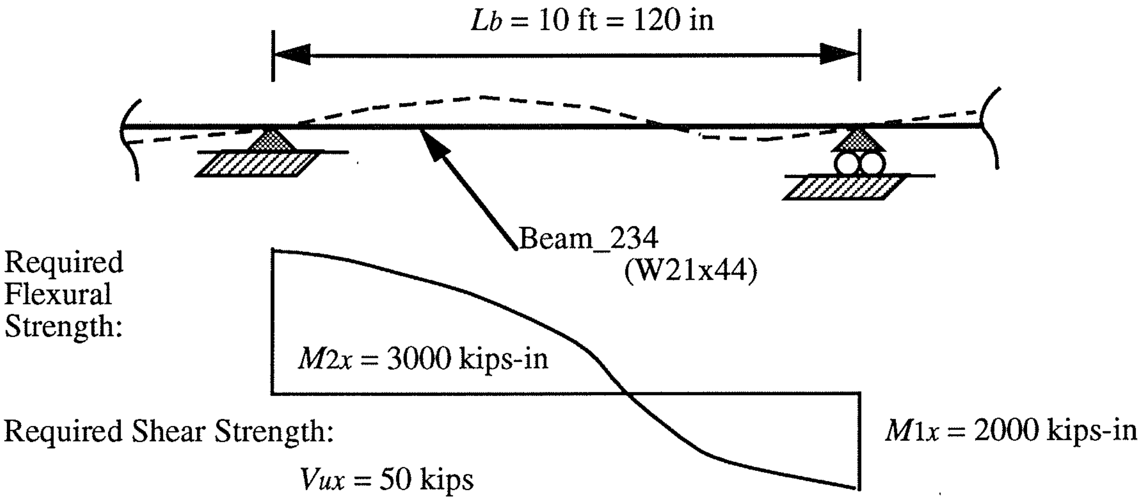
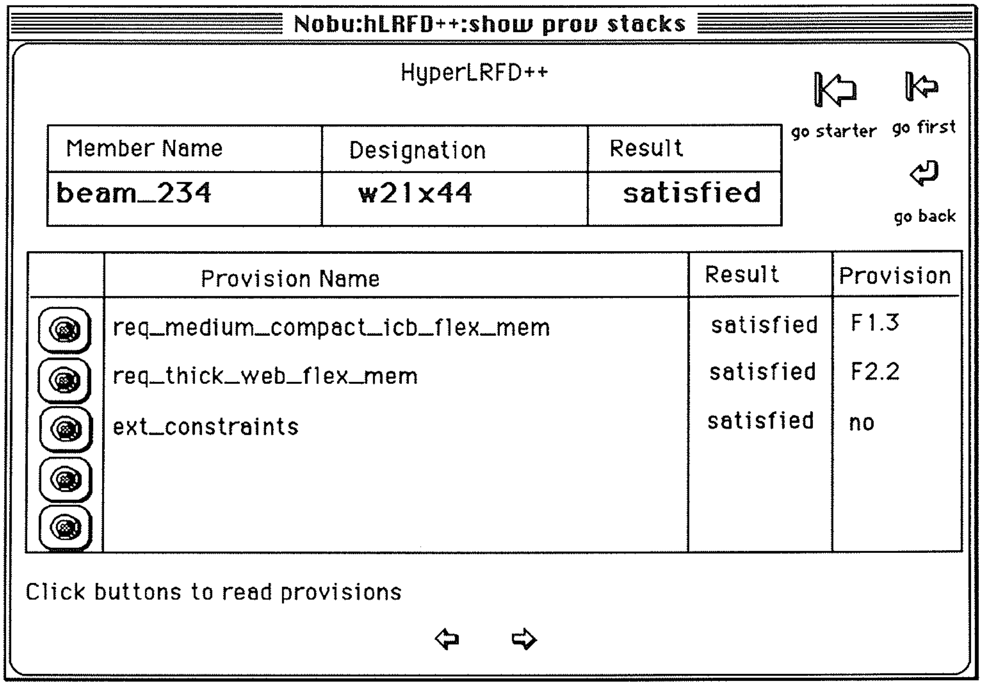


Figure 6-26 Beam_234 and Its Design Condition

Attribute	Value	Default Value	Unit
elastic_modulus	is 29000.	% 29000	ksi
shear_modulus	is 11000.	% 11000	ksi
yield_stress	is 36.	% 36	ksi
tensile_strength	is 58.	% 58	ksi
load_tension	is 0.	% 0	k
load_comp	is 0.	% 0	k
load_moment_x	is 3000.	% 200	k-in
load_moment_y	is 0.	% 0	k-in
load_shear_x	is 50.	% 0	k
load_shear_y	is 0.	% 0	k
load_moment_nt_x	is 0.	% 0	k-in
load_moment_nt_y	is 0.	% 0	k-in
load_moment_lt_x	is 0.	% 0	k-in
load_moment_lt_y	is 0.	% 0	k-in
m1_x	is 2000.	% 0	k-in
m2_x	is 3000.	% 0	k-in
m1_y	is 0.	% 0	k-in
m2_y	is 0.	% 0	k-in
unbraced_length_x	is 120.	% 120	in
unbraced_length_y	is 120.	% 120	in
section_shape	= i_shaped.	% i_shaped	
longitudinal_shape	= prismatic.	% prismatic	
shape	= wshape.	% wshape	
make	= rolled.	% rolled	
braced_x	= no.	% yes	
braced_y	= no.	% yes	
material	= steel.	% steel	
unbraced_cantilever_x	= no.	% no	
unbraced_cantilever_y	= no.	% no	
greater_moment_x	= no.	% no	
greater_moment_y	= no.	% no	
reverse_m1_m2_x	= yes.	% yes	
reverse_m1_m2_y	= yes.	% yes	
have_stiffeners	= no.	% no	
transverse_loading_x	= no.	% no	
transverse_loading_y	= no.	% no	
ends_restrained_x	= no.	% no	
ends_restrained_y	= no.	% no	

```
ext_constraints(Mem, ID, satisfied) :- Mem::bf<-bf(Mem, Bf, ID),
                                        Bf < 10.
ext_constraints(Mem, ID, violated) :- Mem::bf<-bf(Mem, Bf, ID),
                                        Bf >= 10.
```

Figure 6-27 Attributes of Beam_234



Nobu:hLRFD++:show prov stacks

HyperLRFD++

Member Name	Designation	Result
beam_234	w21x44	satisfied

	Provision Name	Result	Provision
◎	req_medium_compact_icb_flex_mem	satisfied	F1.3
◎	req_thick_web_flex_mem	satisfied	F2.2
◎	ext_constraints	satisfied	no
◎			
◎			

Click buttons to read provisions

Figure 6-28 The Result of Checking Beam_234

6.5 Detailed Design of a Flexural Compression Member

This section illustrates the design of a flexural compression member (beam-column) as a part of the frame structure shown in Figure 6-30. The design procedure is divided into two basic steps:

1. Trial member selection: the system selects a trial W shape member.
2. Conformance checking: the system checks the conformance of the member with design provisions.

These two tasks are described in the following subsections.

```
:-   checker<-check(beam_234,  w21x44,  Result)

Preliminary Design or Detail Design? (p/d) = d
classified as steel_mem
classified as flex_mem
classified as prism_flex_mem
classified as icbox_major_flex_mem
classified as compact_icb_flex_mem
classified as medium_compact_icb_flex_mem
classified as has_web_flex_mem_x
classified as thick_web_flex_mem

***   Requirement List = [req_medium_compact_icb_flex_mem,
                          req_thick_web_flex_mem,
                          ext_constraints]

***   Requirement req_medium_compact_icb_flex_mem is being evaluated.
req_medium_compact_icb_flex_mem is SATISFIED.

***   Requirement req_thick_web_flex_mem is being evaluated.
req_thick_web_flex_mem is SATISFIED.

***   Requirement ext_constraints is being evaluated.
ext_constraints are satisfied

***   Final Result is SATISFIED   ***
Do you want to read the provisions? (y/n) : y
Please change to HyperCard by clicking the Apple menu.
Then, click the #4 button of "Show Provisions."
Nº1         Result = satisfied
```

Figure 6-29   The Session of Conformance Checking of "Beam_234"

6.5.1 Selection of a Trial Beam-Column Member

Figure 6-31 summarizes the attribute values of the member "beam_column_45" defined as an instance of the class "steel beam-column" in the Object Model.

In this section the procedure in selecting a trial section for the beam-column employed in the prototype system is described. In the flexural compression member design module, which is a submodule of the Component Design Module, the procedure in selecting a trial member is based on the approximate interactive equation given in the LRFD Manual [AISC 86]:

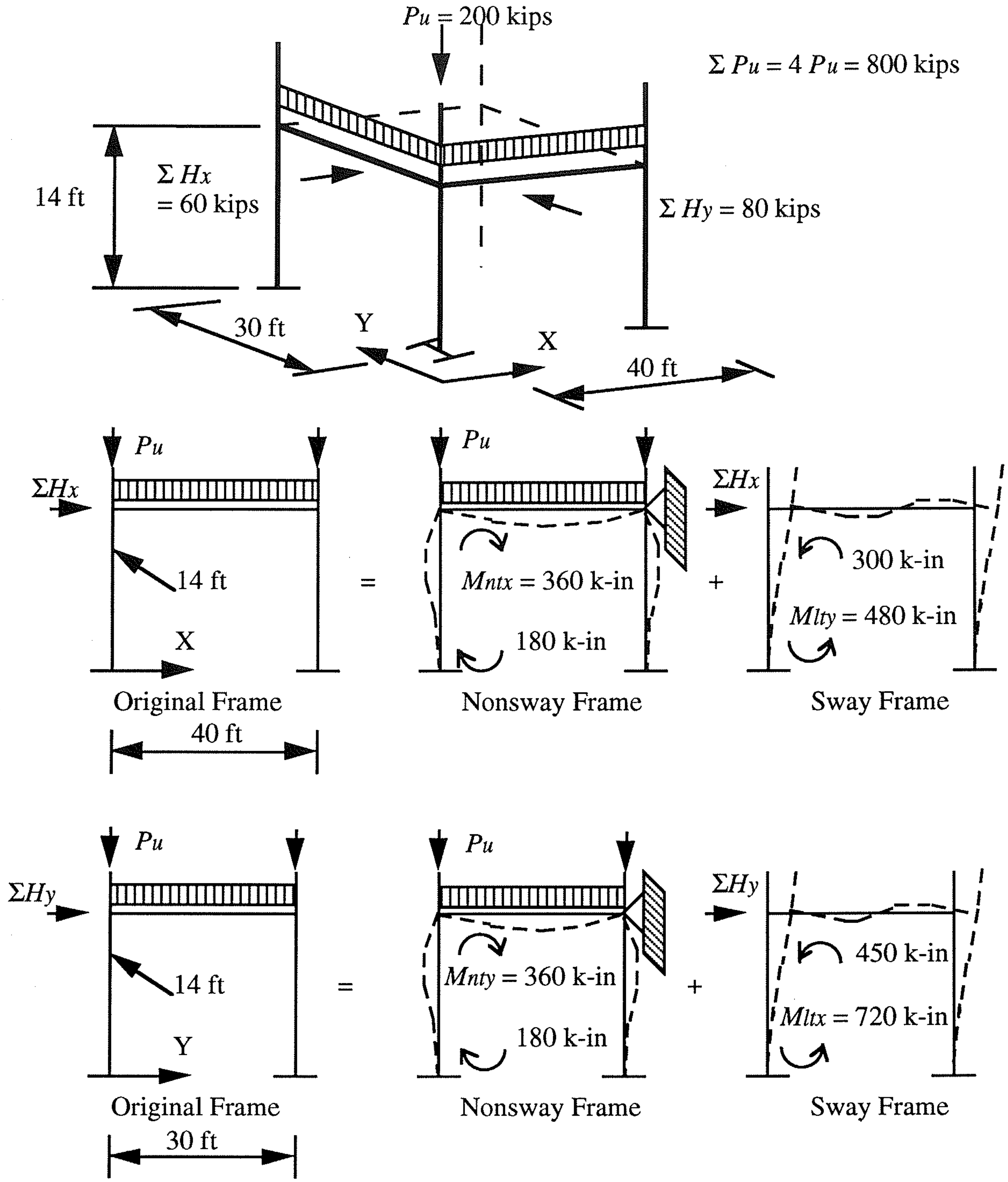


Figure 6-30 The Frame Structure and Load Condition of "Beam_column_45"

Attribute	Value	Default Value	Unit
elastic_modulus	is 29000.	% 29000	ksi
shear_modulus	is 11000.	% 11000	ksi
yield_stress	is 36.	% 36	ksi
load_tension	is 0.	% 0	k
load_comp	is 200.	% 300	k
load_moment_x	is unknown.	% 0	k
load_moment_y	is unknown.	% 0	k
load_shear_x	is 0.	% 0	k
load_shear_y	is 0.	% 0	k
load_moment_nt_x	is 360.	% 0	k-in
load_moment_nt_y	is 360.	% 0	k-in
load_moment_lt_x	is 720.	% 0	k-in
load_moment_lt_y	is 480.	% 0	k-in
m1_x	is 180.	% 0	k-in
m2_x	is 360.	% 0	k-in
m1_y	is 180.	% 0	k-in
m2_y	is 360.	% 0	k-in
unbraced_length_x	is 168.	% 120	in
unbraced_length_y	is 168.	% 120	in
effective_length_factor_x	is unknown.	% 1	
effective_length_factor_y	is unknown.	% 1	
effective_length_factor_z	is 1.	% 1	
effective_length_factor_x_nt	is 1.	% 1	
effective_length_factor_y_nt	is 1.	% 1	
effective_length_factor_x_lt	is 1.2.	% 1.2	
effective_length_factor_y_lt	is 1.2.	% 1.2	
sum_load_comp	is 800.	% 640	k
translation_deflection_x	is 0.336.	% 2.4	in
translation_deflection_y	is 0.336.	% 2.4	in
number_of_columns_in_story	is unknown.	% 8	
sum_horizontal_forces_x	is 60.	% 100	k
sum_horizontal_forces_y	is 80.	% 100	k
stiffener_distance	is unknown.	% 10	in
section_shape	= i_shaped.	% i_shaped	
longitudinal_shape	= prismatic.	% prismatic	
shape	= wshape.	% wshape	
make	= rolled.	% rolled	
braced_x	= no.	% yes	
braced_y	= no.	% yes	
material	= steel.	% steel	
unbraced_cantilever_x	= no.	% no	
unbraced_cantilever_y	= no.	% no	
greater_moment_x	= no.	% no	
greater_moment_y	= no.	% no	
reverse_m1_m2_x	= yes.	% yes	
reverse_m1_m2_y	= yes.	% yes	
have_stiffeners	= no.	% no	
transverse_loading_x	= no.	% no	
transverse_loading_y	= no.	% no	
ends_restrained_x	= unknown.	% no	
ends_restrained_y	= unknown.	% no	
mu_x_given	= unknown.	% unknown	
mu_y_given	= unknown.	% unknown	

Figure 6-31 Attributes and Their Values of "Beam_column_45"

$$P_{u,eff} = P_u + M_{ux}\, m + M_{uy}\, mU \tag{1}$$

where

$P_{u,eff}$ = effective axial load,

P_u = required compressive strength, kips

M_{ux}, M_{uy} = required flexural strength, kip-in

m and U are factors determined from Table 6-2 [AISC 86] [Rokach 91],

Once $P_{u,eff}$ has been obtained, the beam-column is designed as a compression member whose required compressive strength is equal to $P_{u,eff}$. With the effective axial force, a candidate W shape member can be selected as a trial section by the compression member design heuristics described in Section 6.2.2. To calculate the effective axial load, the values of M_{ux}, M_{uy} , m, and U are necessary.

The LRFD manual [AISC 86] provides a table to establish the values of m and U, which is depicted in Table 6-2. In HyperLRFD++, the table is written in Prolog++ sentences as:

```
flex_comp_list(w14,KyLy,2.2,1.5) :- KyLy =< 10.
flex_comp_list(w14,KyLy,2.0,1.5) :- KyLy > 10.
```

To determine the coefficients m and U, the user needs to select one designation group from the list [W14, W12, W10, W8] and input the effective length factor K_x, For this example, W14 section and K_x = 1.2 are used in the calculation. Based on this information, the system calculates the effective length about the x-axis, K_xL_x = 1.2 * 14 = 16.8 ft, and retrieves the values of m and U as 2.0 and 1.5.

To estimate the required flexural strength M_{ux}, the following equation can be used [AISC 86]:

$$M_{ux} = B_{1x}\, M_{ntx} + B_{2x}\, M_{ltx} \tag{2}$$

where

M_{ntx} = required flexural strength about the x-axis in member assuming there is no lateral translation of the frame, kip-in.

M_{ltx} = required flexural strength about the x-axis in member as a result of lateral translation of the frame only, kip-in

$$B_{1x} = \frac{C_{mx}}{\left(1 - \dfrac{P_u}{P_{ex,nt}}\right)} \geq 1 \tag{3}$$

Table 6-2 Table for Determining m and U for Beam-Columns: Fy = 36 ksi (Adapted from [AISC 86])

	m							U
KL, ft	10	12	14	16	18	20	≥ 22	
W8	3.6	3.5	3.4	3.1	2.8	2.4	2.4	1.5
W10	3.1	3.0	3.0	2.9	2.8	2.5	2.4	1.5
W12	2.5	2.5	2.4	2.4	2.4	2.4	2.4	1.5
W14	2.2	2.0	2.0	2.0	2.0	2.0	2.4	1.5

$$B_{2x} = \frac{1}{1 - \sum P_u \left(\frac{\Delta_{ohy}}{\sum H_y L_x} \right)} \tag{4}$$

or

$$B_{2x} = \frac{1}{1 - \frac{\sum P_u}{\sum P_{ex,lt}}} \tag{5}$$

C_{mx} = coefficient for beam-column,

$P_{ex,nt} = \pi^2 EI_x / (K_{x,nt} L_x)$ where $K_{x,nt} \leq 1.0$ (6)

$K_{x,nt}$ = effective length factor about the x-axis in no lateral translation case,

ΣP_u = required axial strength of all columns in a story, kips

Δ_{ohy} = translation deflection of the story under consideration, in

ΣH_y = sum of all story horizontal forces producing Δ_{ohy}, kips

$P_{ex,lt} = \pi^2 EI_x / (K_{x,lt} L_x)$ where $K_{x,lt} \geq 1.0$ (7)

$K_{x,lt}$ = effective length factor about the x-axis in lateral translation case,

The system first assumes B_{1x} to be 1.0 and calculates B_{2x} based on Equation (4), since Equation (5) requires the member properties which are yet to be determined. The story drift index, Δ_{ohy} / L_y, especially for tall buildings, is a design criterion [Rokach 91], which is assumed to be 1/500 by the user in this example. The attribute "translation_ deflection_x" can be estimated to be:

```
:-  designer<-design(beam_column_45,  ID)

*****  TRIAL SECTION SELECTION  *****
Enter mu_x_given (default is no :enter d) = d
no
Enter mu_y_given (default is no :enter d) = d
no
Want to design a flex_comp_mem as a comp_mem or a flex_mem
            or a flex_comp_mem?
Enter (c:comp_mem/f:flex_mem/fc:flex_comp_mem) : fc
Designing wshape_comp_mem
Enter a designation (e.g., default - w14, enter d) : d
ColumnClass = w14_36_columns
To calculate B2_x, use (1)translation deflection,
                    or (2)sum of axial loads in columns. 1 or 2:1

B2_x = 1.020408163265306122
Mu_x = 1094.693877551020408
To calculate B2_y, use (1)translation deflection,
                    or (2)sum of axial loads in columns. 1 or 2:1

B2_y = 1.027397260273972603
Mu_y = 853.1506849315068494
Pu_eff = 595.7366507523405812
TRIAL SECTION ID = w14x90
```

Figure 6-32 The Session of Trial Section Selection of Designing "Beam_column_45"

$$\Delta_{ohy} = 14 * 12 / 500 = 0.336 \text{ (in)}$$

The system calculates:

$$B_{2x} = 1 / (1 - 800 / 80 * (0.336 / (14 * 12))) = 1.020$$

and by Equation (2)

$$M_{ux} = B_{1x} M_{ntx} + B_{2x} M_{ltx} = 1.0 * 360 + 1.020 * 720 = 1094.4 \text{ (k-in)}$$

M_{uy} can be determined similarly as 853.0 (k-in) by the system.

Thus, following Equation (1), the system determines the effective axial load as:

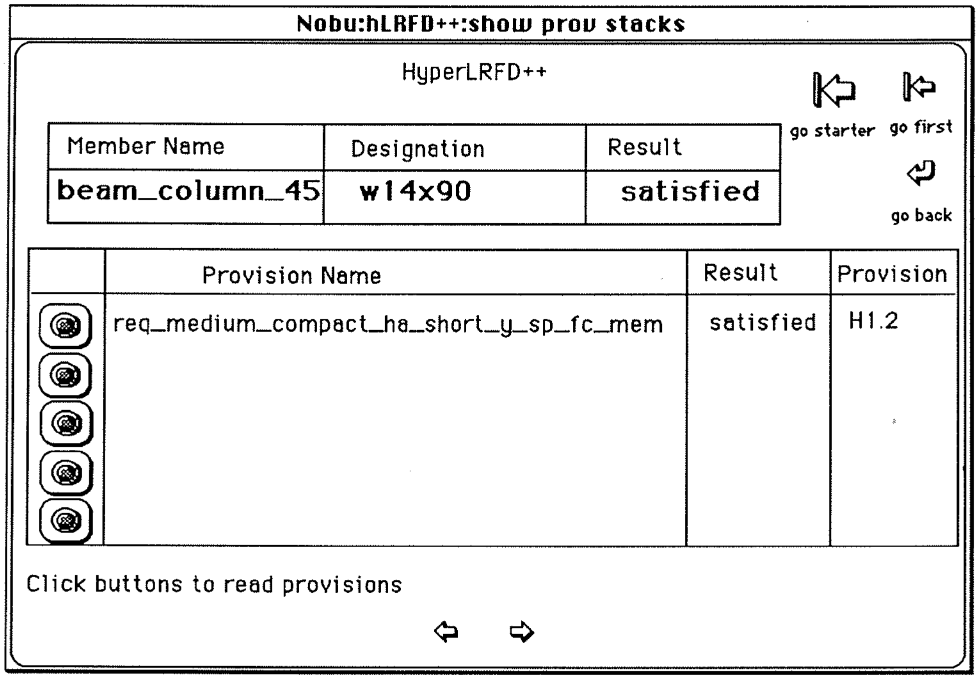
$$P_{u,eff} = 200 + 2.0 * 1094.4 / 12 + 2.0 * 1.5 * 853.0 / 12 = 595.65$$

Given the values $P_{u,eff}$ and $K_yL_y = 1.2 * 14.0 = 16.8$ ft , the candidate member W14x90 can now be selected from Table 6-1. This trial design can now be checked for conformance with design provisions. This process is summarized in the session as shown in Figure 6-32.

6.5.2 Conformance Checking

Next, the system checks whether the selected W14x90 member satisfies all applicable requirements. By traversal of the Member Class Hierarchy, the member is classified as the leaf node class of "medium_compact_ha_short_y_sp_fc_mem." The corresponding requirement "req_medium_compact_ha_short_y_sp_fc_mem" needs to be checked. In conformance checking, the system uses Equation (5) to determine B_{2x} because $P_{ex,lt}$ can be calculated with the given section designation, while the true value Δ_{ohx} in Equation (4) requires a detailed structural analysis. The result is tabulated as shown in Figure 6-33 and the design checking session is given in Figure 6-34. The design report is shown in Figure 6-35.

In this example, W14x90 fulfills the requirement; otherwise the Component Design Module will proceed to select the next heavier section, i.e., W14x99, as a candidate member and the design procedure will be repeated.



HyperLRFD++

Member Name	Designation	Result
beam_column_45	w14x90	satisfied

	Provision Name	Result	Provision
	req_medium_compact_ha_short_y_sp_fc_mem	satisfied	H1.2

Click buttons to read provisions

Figure 6-33 The Result of Designing Beam_column_45

```
*****  CHECKING THE TRIAL SECTION  *****
Preliminary Design or Detail Design? (p/d) = d
***  Traversing the hierarchy from the selected class:
classified as steel_mem
classified as flex_comp_mem
classified as prism_flex_comp_mem
classified as sym_p_fc_mem
classified as non_slender_sp_fc_mem
classified as non_slender_y_sp_fc_mem
classified as short_ns_y_sp_fc_mem
classified as high_axial_short_ns_y_sp_fc_mem
classified as compact_ha_short_y_sp_fc_mem
classified as medium_compact_ha_short_y_sp_fc_mem

***  Requirement List = [req_medium_compact_ha_short_y_sp_fc_mem]

***  Requirement req_medium_compact_ha_short_y_sp_fc_mem is being
evaluated.

To calculate B2_x, use (1)translation deflection,
                    or (2)sum of axial loads in columns. 1 or 2:2
B2_x = 1.029258715358502572l
Mu_x = 1101.1387489641808453
To calculate B2_y, use (1)translation deflection,
                    or (2)sum of axial loads in columns. 1 or 2:2
B2_y = 1.0599252814229708757
Mu_y = 880.83780284957876744
Equation Value = 0.787370284904902989

req_medium_compact_ha_short_y_sp_fc_mem is SATISFIED.

Do you want to read the provisions? (y/n) : y
Please change to HyperCard by clicking the Apple menu.
Then, click the #4 button of "Show Provisions."

***  RESULT  ***
*****  w14x90  is the design  *****
Nº1         ID = w14x90
```

Figure 6-34 The Session of Conformance Checking of Designing "Beam_column_45" (Continued from Figure 6-32)

```
1. Problem Description

Task:         component design
Member Name:  beam_column_45

   Attribute                            Value             Unit

elastic_modulus                       is 29000.          ksi
shear_modulus                         is 11000.          ksi
yield_stress                          is 36.             ksi
load_tension                          is 0.              k
load_comp                             is 200.            k
load_shear_x                          is 0.              k
load_shear_y                          is 0.              k
load_moment_nt_x                      is 360             k-in
load_moment_nt_y                      is 360             k-in
load_moment_lt_x                      is 720             k-in
load_moment_lt_y                      is 480             k-in
m1_x                                  is 180             k-in
m2_x                                  is 360             k-in
m1_y                                  is 180             k-in
m2_y                                  is 360             k-in
unbraced_length_x                     is 168.            in
unbraced_length_y                     is 168.            in
effective_length_factor_x_nt          is 1.
effective_length_factor_y_nt          is 1.
effective_length_factor_x_lt          is 1.2.
effective_length_factor_y_lt          is 1.2.
sum_load_comp                         is 800.            k
translation_deflection_x              is 0.336.          in
translation_deflection_y              is 0.336.          in

      (The rest is omitted.)

2. Design Strategy

Detailed design

3. Result
   Designation: w14x90
   Requirement                                    Result
   req_medium_compact_ha_short_y_sp_fc_mem        satisfied
   (requirement for a compact high axial compression symmetric prismatic
    flexural compression member which y-axis inelastic buckling governs)
```

Figure 6-35 Design Report for the Member "Beam_column_45"

6.6 Summary

In this chapter the two modules of the Object-Logic model for conformance checking and component design were described. A few illustrative examples were provided to demonstrate the Hyper-Object-Logic model to conformance checking and component design. The first example was to perform preliminary design of a column. The heuristics for selecting a plausible W shape section candidate for a column was described in detail. The second example was to perform a detailed checking on the column designed in the first example. This example showed how the user can utilize provisions, explanations, background information, and external programs in the HyperDocument system to facilitate the re-design process. The third example was to check a flexural member in the detailed design phase. The fourth example was to design a beam-column in the detailed design phase. The heuristics of selecting a plausible W shape section candidate for a beam-column was described in detail. These examples of using HyperLRFD++ have shown the feasibility and practicality of the Hyper-Object-Logic model for conformance checking and component design applications.

Chapter 7
Standards Analysis

In this chapter, the Standards Analysis Module is described. The Standards Analysis Module checks the completeness, uniqueness, and correctness of the standard at the provision level and the organization level. It also checks whether the dependency network representing the relationships among the Method Objects is connected and acyclic. Section 7.1 explains how the module checks the rules representing an individual provision. Section 7.2 describes how the standards organization can be checked by the module. In Section 7.3, the procedure for checking the relations among provisions is described. For each section, examples are given to illustrate the analysis module.

7.1 Analysis of Individual Provisions

The Method Object consists of a set of Object-Logic sentences for determining its corresponding data item. Analysis of a set of sentences of the Method Object consists of checking the provisions of completeness, uniqueness, and correctness. The first two are syntactic properties, while correctness is a semantic property. Completeness of a set of rules in the Method Object ensures that all possible conditions are covered. Uniqueness ensures that only one rule is applicable for any given condition. Uniqueness is composed of lack of redundancy and lack of contradiction. A set of rules is said to be redundant if more than one rule in the same rule group is applicable for a given condition. A set of rules is said to be in contradiction if different actions are suggested by multiple rules in the same rule group that are applicable for a given condition. Correctness ensures that a set of sentences in the Method Object represents the meaning, intentions, and implications of the corresponding provision correctly. Checking these properties of requirement and determinant Method Objects is performed at the provision level.

For checking syntactic completeness and uniqueness, this thesis adopts, modifies, and implements the approach proposed by Jain et al. [Jain 89]. The semantic correctness of the Method Object can be checked by comparing the rules in the Method Object and its corresponding provision stored in the Document Base of the HyperDocument system. The pointers between Method Objects and provision documents facilitate comparing these two types of documents. This procedure is discussed in Section 7.1.5.

The basic procedure for checking the completeness and uniqueness of a provision stored as Method Object is shown in Figure 7-1. The Standards Analysis Module parses the Object-Logic sentences of the Method Object and formulates them as sentences. These sentences are then proved for completeness and uniqueness (lack of redundancy and lack of contradiction) using the resolution principle.

Each Method Object is represented as a group of Object-Logic sentences written in the form:

A_i :- C_i.

where

$i = 1, 2, 3, \ldots\ldots\ldots, n,$

n = total number of sentences in a group,

A_i : action or conclusion part of the sentence,

:- : equivalent to logical ←

C_i : conjunction of conditions, i.e., $C_i = C_{i1}, C_{i2}, C_{i3}, \ldots\ldots\ldots, C_{ik_i},$

$k_i \geq 0.$

The conditions C_{ij}'s are divided into two groups:

- conditions which represent the logic of the provision, and
- conditions which are not directly related to the provision, such as "write(Fy)" and "input_attr(Mem, yield_strength, 36)."

During the parsing of the Object-Logic sentences, the latter type conditions are ignored and removed since they are not related to the checking of completeness and uniqueness.

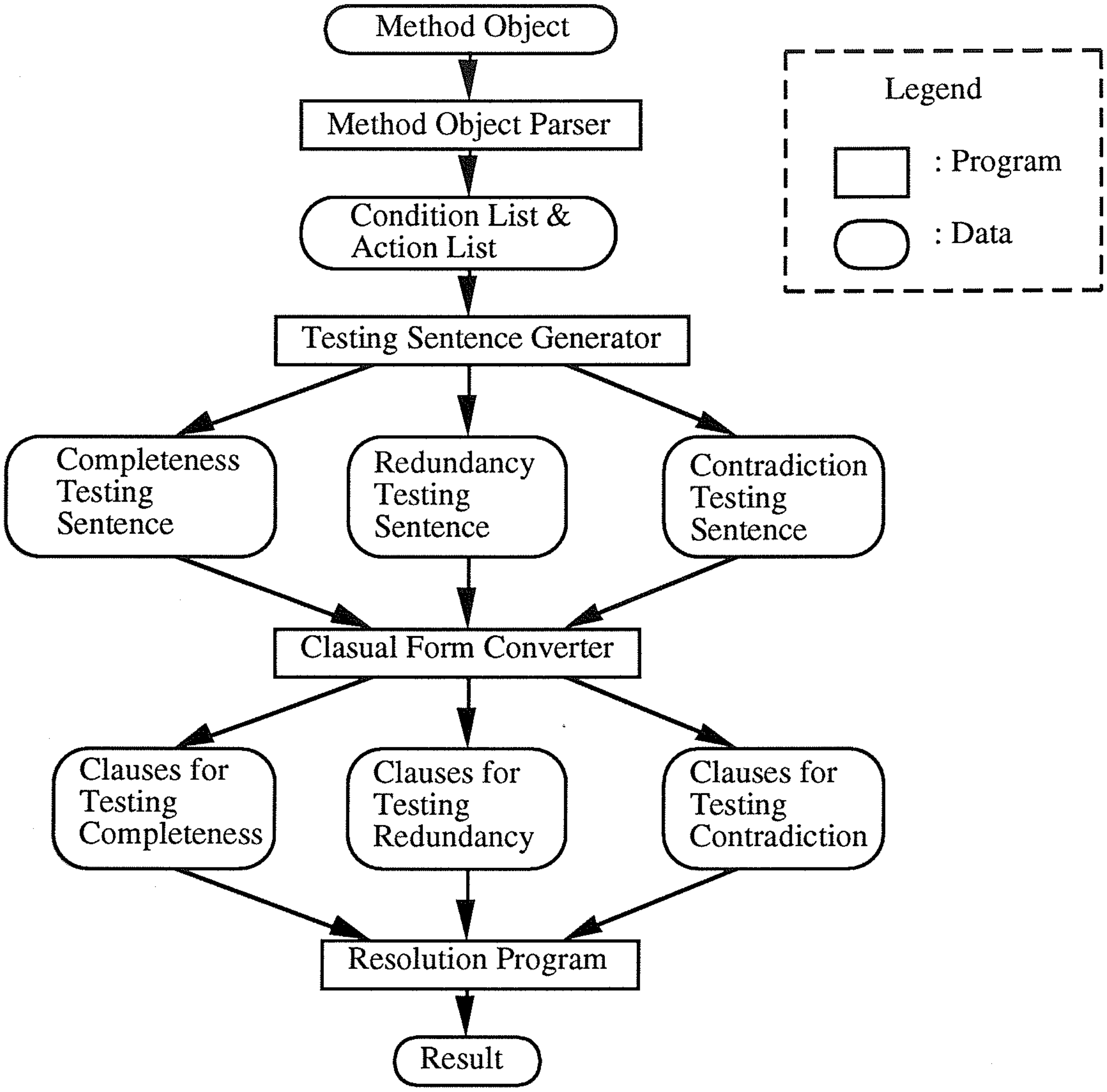

Method Object
Method Object Parser
Condition List & Action List
Testing Sentence Generator
Completeness Testing Sentence
Redundancy Testing Sentence
Contradiction Testing Sentence
Clasual Form Converter
Clauses for Testing Completeness
Clauses for Testing Redundancy
Clauses for Testing Contradiction
Resolution Program
Result
Legend
: Program
: Data

Figure 7-1 The Basic Procedure for Checking Individual Provisions

Completeness, lack of redundancy, and lack of contradiction can be verified by proving the validity of the following sentences of each Method Object [Jain 89]:

1. Completeness:

$$\bigvee_{i=1}^{n} C_i$$

2. Lack of redundancy:

$$\bigwedge_{i=1,\, j=i+1}^{i=n\text{-}1,\, j=n} (\neg C_i \vee \neg C_j)$$

3. Lack of Contradiction:

$$\bigwedge_{i=1,\, j=i+1}^{i=n\text{-}1,\, j=n} [\neg C_i \vee \neg C_j \vee (A_i = A_j)]$$

The procedure that proves the validity of each of above sentences consists of the following steps:

1. Negate the testing sentence,
2. Convert it to the clausal form,
3. Perform resolution on the resulting clauses to deduce the empty clause { }.
4. If the empty clause is deduced, the sentence is proved to be valid; otherwise, the sentence is not valid.

Since checking the three properties does not require the unification of variables appearing in the sentences, the three sentences above can be treated as propositional logic sentences although the sentences may contain variables. For example, a condition:

```
    Mem::gross_area<-gross_area(Mem,Ag,ID)
```

is treated as a one-word condition in propositional logic. Resolution in propositional logic is deterministic. That is, one can tell whether the empty clause can be deduced or not by exhaustive search. In propositional logic, the basic procedure in converting a propositional logic sentence to clausal form is as follows:

1. Eliminate implication by converting a form of A :- C into $A \vee \neg C$,
2. Distribute negations over the other logical operators by De Morgan's laws,

$$\neg (X \wedge Y) \equiv \neg X \vee \neg Y$$

$$\neg (X \vee Y) \equiv \neg X \wedge \neg Y$$

3. Put the sentence into conjunctive normal form, i.e., a conjunction of disjunctions of conditions by the distributivity law,

$$(X \wedge Y) \vee Z \equiv (X \vee Z) \wedge (Y \vee Z)$$

4. Write the conjunction obtained in the above step as a set of clauses.

The procedures for checking completeness, lack of redundancy, and lack of contradiction are described in the following subsections.

7.1.1 Checking Completeness

As noted earlier, the completeness of the rules in a Method Object can be checked by the following sentence:

$$\bigvee_{i=1}^{n} C_i$$

By negating the sentence, we have:

$$\neg\left(\bigvee_{i=1}^{n} C_i\right)$$

$$\equiv \bigwedge_{i=1}^{n} \neg C_i$$

$$\equiv \neg(C_{11} \wedge C_{12} \wedge \;..........\; \wedge C_{1k_1}) \wedge \neg(C_{21} \wedge C_{22} \wedge \;..........\; \wedge C_{2k_2}) \wedge$$
$$.................... \wedge \neg(C_{n1} \wedge C_{n2} \wedge \;.............\; \wedge C_{nk_1})$$

$$\equiv (\neg C_{11} \vee \neg C_{12} \vee \;........\; \vee \neg C_{1k_1}) \wedge (\neg C_{21} \vee \neg C_{22} \vee \;........\; \vee \neg C_{2k_2}) \wedge$$
$$.................... \wedge (\neg C_{n1} \vee \neg C_{n2} \vee \;.............\; \vee \neg C_{nk_1})$$

This sentence can be converted into the clausal form as:

$\{\neg C_{11}, \neg C_{12}, \;.............\;, \neg C_{1k_1}\}$

$\{\neg C_{21}, \neg C_{22}, \;.............\;, \neg C_{2k_2}\}$

....

$\{\neg C_{n1}, \neg C_{n2}, \ ...., \neg C_{nk_n}\}$

The approach for checking completeness is to deduce an empty clause from the above clauses by resolution [Jain 89].

Unfortunately, unless the sentences are directly translated from a limited-entry decision table, the procedure described above is not sufficient. For example, the following three sentences about a section's property:

```
section(X) :- singly_symmetric(X).
section(X) :- doubly_symmetric(X).
section(X) :- unsymmetric(X).
```

are a complete set. Applying the procedure in checking for completeness gives:

```
singly_symmetric(X) ∨ doubly_symmetric(X) ∨ unsymmetric(X).
```

Negating this sentence, we get:

```
¬ singly_symmetric(X) ∧ ¬ doubly_symmetric(X) ∧ ¬ unsymmetric(X).
```

This sentence can be converted into the clausal form as:

```
{¬ singly_symmetric(X), ¬ doubly_symmetric(X), ¬ unsymmetric(X)} (i)
```

The above clause cannot be deduced to the empty clause by resolution although the three sentences about a section property are complete.

This ambiguous problem can be resolved by adding a dictionary of complete combinations to the set of clauses before performing the resolution procedure. The dictionary contains, for example, the following sentence:

```
singly_symmetric(X) ∨ doubly_symmetric(X) ∨ unsymmetric(X).
```

This sentence is converted into the clausal form of:

```
{singly_symmetric(X)}                                  (ii)
{doubly_symmetric(X)}                                  (iii)
{unsymmetric(X)}.                                      (iv)
```

Resolution on the clauses from (i) through (iv) now deduces an empty clause.

Let us now consider an example provision from Section F1.3 for flexural members of the AISC LRFD specification [AISC 86], which describes the coefficient C_b, a factor to account for moment gradient in beam strength [Salmon 90]. The provision is shown in

C_b = $1.75 + 1.05\,(M_1 / M_2) + 0.3\,(M_1 / M_2)^2 \leq 2.3$ where M_1 is the smaller and M_2 is the larger end moment in the unbraced segment of the beam; M_1 / M_2 is positive when the moments cause reverse curvature and negative when bent in single curvature.

C_b = 1.0 for unbraced cantilevers and for members where the moment within a significant portion of the unbraced segment is greater than or equal to the larger of the segment end moments.

Figure 7-2 The Provision about the Coefficient C_b

```
cb(Mem,1,Id):-
   self<-input_attr(Mem,unbraced_cantilever_y,no),
   Mem::unbraced_cantilever_y = yes.

cb(Mem,1,Id):-
   self<-input_attr(Mem,greater_moment_x,no),
   Mem::greater_moment_x = yes.

cb(Mem,Cb,Id):-
   not(Mem::unbraced_cantilever_y = yes),
   not(Mem::greater_moment_x = yes),
   Cb_t is 1.75 + 1.05 * Mem::m1_x / Mem::m2_x + 0.3 * (Mem::m1_x /
                Mem::m2_x)^2,
   self<-min(Cb_t,2.3,Cb).
```

Figure 7-3 Object-Logic Sentences in the Method Object "Cb"

Figure 7-2. The Method Object "cb" represents the provision about C_b and is shown in Figure 7-3. To check the completeness of a specific provision in HyperLRFD++, the system first parses the Method Object and obtains the condition and action lists. The Standards Analysis Module eliminates the unrelated conditions such as procedures for input attribute values and arithmetic expressions. In this example, the condition and action lists, after elimination of the unrelated sentences, are:

```
ConditionList =
        [[Mem::unbraced_cantilever_y=yes],
         [Mem::greater_moment_x=yes],
         [not (Mem::unbraced_cantilever_y=yes),
          not (Mem::greater_moment_x=yes)]]

ActionList =
        [cb(Mem,1,Id), cb(Mem,1,Id), cb(Mem,Cb_1,Id)]
```

The resulting conditions are:

C_1: Mem::unbraced_cantilever_x = yes,

C_2: Mem::greater_moment_y = yes,

C_3: not(Mem::unbraced_cantilever_x = yes),
not(Mem::greater_moment_y = yes).

Let us denote the conditions as:

C_1: U,

C_2: G,

C_3: $\neg U, \neg G$.

By substituting the conditions into the testing sentence for completeness, we have:

$$U \vee G \vee (\neg U \wedge \neg G)$$

Then, the system automatically negates the sentence:

$$\neg [U \vee G \vee (\neg U \wedge \neg G)]$$

By converting this sentence into clausal form, we obtain:

$\{ \neg U \}$ (a)

$\{ \neg G \}$ (b)

$\{ U, G \}$ (c)

By performing resolution on these clauses, we obtain:

$\{ G \}$ from (a) and (c) (d)

$\{ \}$ from (b) and (d) (e)

which is an empty clause. Therefore, the rules in the Method Object "cb" are complete. The session of checking completeness of the Method Object "cb" using HyperLRFD++ is shown in Figure 7-4.

```
:-  completeness

ConditionList =
[[_1412::unbraced_cantilever_y=yes], [_1412::greater_moment_x=yes],
 [not (_1412::unbraced_cantilever_y=yes),
  not (_1412::greater_moment_x=yes)]]

NegatedTestingSentence =
not (_1747::unbraced_cantilever_y=yes v _1747::greater_moment_x=yes v
not (_1747::unbraced_cantilever_y=yes)&
not (_1747::greater_moment_x=yes))

CLAUSES
   *** clause2(not (_1747::unbraced_cantilever_y=yes))
   *** clause2(not (_1747::greater_moment_x=yes))
   *** clause2(_1747::unbraced_cantilever_y=yes v
               _1747::greater_moment_x=yes)

PERFORMING RESOLUTION
_1747::greater_moment_x=yes
[] : Empty clause found
GOOD!  The given rules are COMPLETE.
Nº1         yes
```

where & : logical "and"
v : logical "or"
`clause2` : a predicate to indicate an asserted clause
four digit number following "_" such as "_1412" indicates a variable ID in Prolog.

Figure 7-4 The Session of Checking Completeness

7.1.2 Checking Lack of Redundancy

The redundancy among the rules in a Method Object can be checked by the following sentence:

$$\bigwedge_{i=1,\, j=i+1}^{i=n\text{-}1,\, j=n} (\neg C_i \vee \neg C_j)$$

By negating the sentence, we have:

$$\neg\, [\bigwedge_{i=1,\, j=i+1}^{i=n\text{-}1,\, j=n} (\neg C_i \vee \neg C_j)]$$

$$\equiv \bigvee_{i=1,\, j=i+1}^{i=n\text{-}1,\, j=n} (C_i \wedge C_j)$$

$$\begin{aligned}
\equiv (C_1 \wedge C_2) \vee (C_1 \wedge C_3) \vee \; & \ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \vee (C_1 \wedge C_n) \vee \\
\vee (C_2 \wedge C_3) \vee \; & \ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \vee (C_2 \wedge C_n) \vee \\
& \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
& \qquad\qquad\qquad\quad \vee (C_{n\text{-}1} \vee C_n)
\end{aligned}$$

$$\begin{aligned}
\equiv\; & [(C_{11} \wedge C_{12} \wedge \;\ldots\ldots\ldots\; \wedge C_{1k_1}) \wedge (C_{21} \wedge C_{22} \wedge \;\ldots\ldots\ldots\; \wedge C_{2k_2})] \vee \\
& \quad \ldots\ldots\ldots\ldots\ldots\ldots \\
& \vee [(C_{n\text{-}1,1} \wedge C_{n\text{-}1,2} \wedge \ldots\ldots \wedge C_{n\text{-}1,k_{n\text{-}1}}) \wedge (C_{n1} \wedge C_{n2} \wedge \ldots\ldots \wedge C_{nk_n})]
\end{aligned}$$

The above sentence can be converted into conjunctive normal form by the distributivity law, and furthermore, converted into a set of clauses. The converted clauses can be tested for redundancy using the resolution principle.

Let us now consider the set of rules for the Method Object "cb" demonstrated in the completeness checking in the previous section. The negated sentence for these rules is:

$$(U \wedge G) \vee [U \wedge (\neg U \wedge \neg G)] \vee [G \wedge (\neg U \wedge \neg G)]$$

Converting this sentence into clausal form, we have:

{ U, G } (a)
{ U, ¬ G } (b)
{ G, ¬ U } (c)

By performing resolution on these clauses, we obtain

{ U } from (a) and (b) (d)
{ G } from (b) and (c) (e)

which is not an empty clause. Thus, the rules in the Method Object "cb" are redundant. The reason that the rules are redundant is that if both U and G are true, i.e., the member has the attribute values:

```
Mem::unbraced_cantilever_y = yes, and
Mem::greater_moment_x = yes,
```

both first and second rules are applicable. However, in the ordinary unbraced cantilever, it is unlikely that the moment within a significant portion of the unbraced segment is greater than the end moment. Thus, in practical point of view, this redundancy is not a problem.

```
:-  redundancy

ConditionList =
[[_1133::unbraced_cantilever_y=yes], [_1133::greater_moment_x=yes],
 [not (_1133::unbraced_cantilever_y=yes),
  not (_1133::greater_moment_x=yes)]]

NegatedTestingSentence =
_1614::unbraced_cantilever_y=yes&_1614::greater_moment_x=yes v
_1614::unbraced_cantilever_y=yes&not (_1614::unbraced_cantilever_y=yes)&
not (_1614::greater_moment_x=yes)v _1614::greater_moment_x=yes&
not (_1614::unbraced_cantilever_y=yes)&
not (_1614::greater_moment_x=yes)v _1614::greater_moment_x=yes&
not (_1614::unbraced_cantilever_y=yes)&not (_1614::greater_moment_x=yes)

CLAUSES
  *** clause2(_1614::unbraced_cantilever_y=yes v
              _1614::greater_moment_x=yes)
  *** clause2(_1614::unbraced_cantilever_y=yes v
              not (_1614::greater_moment_x=yes))
  *** clause2(_1614::greater_moment_x=yes v
              not (_1614::unbraced_cantilever_y=yes))

PERFORMING RESOLUTION
_3647::greater_moment_x=yes v _3647::greater_moment_x=yes
_4355::unbraced_cantilever_y=yes
_4655::unbraced_cantilever_y=yes v _4655::unbraced_cantilever_y=yes
not (_5492::greater_moment_x=yes)v _5492::greater_moment_x=yes
not (_5825::unbraced_cantilever_y=yes)v _5825::unbraced_cantilever_y=yes
_6158::greater_moment_x=yes v _6158::unbraced_cantilever_y=yes
not (_6485::greater_moment_x=yes)v _6485::unbraced_cantilever_y=yes
_6815::unbraced_cantilever_y=yes v _6815::unbraced_cantilever_y=yes
not (_7976::unbraced_cantilever_y=yes)v _7976::greater_moment_x=yes
_8306::greater_moment_x=yes v _8306::greater_moment_x=yes
_9575::unbraced_cantilever_y=yes v _9575::unbraced_cantilever_y=yes
_10892::unbraced_cantilever_y=yes v _10892::unbraced_cantilever_y=yes
_12245::greater_moment_x=yes v _12245::greater_moment_x=yes
_13646::greater_moment_x=yes v _13646::greater_moment_x=yes

Empty clause NOT found
NOT GOOD! The given rules are REDUNDANT.
Nº1         yes
```

Figure 7-5 The Session of Checking Lack of Redundancy

The session of checking lack of redundancy of the Method Object "cb" using HyperLRFD++ is shown in Figure 7-5.

7.1.3 Checking Lack of Contradiction

Contradictory rules which produce multiple different actions for a given set of conditions can be checked using the following sentence:

$$\bigwedge_{i=1,\, j=i+1}^{i=n-1,\, j=n} [\neg C_i \vee \neg C_j \vee (A_i = A_j)]$$

By negating the sentence, we have:

$$\neg(\bigwedge_{i=1,\, j=i+1}^{i=n-1,\, j=n} [\neg C_i \vee \neg C_j \vee (A_i = A_j)])$$

$$\equiv \bigvee_{i=1,\, j=i+1}^{i=n-1,\, j=n} [C_i \wedge C_j \wedge \neg(A_i = A_j)]$$

If $A_i = A_j$ is true:

$$[C_i \wedge C_j \wedge \neg(A_i = A_j)]$$

$$\equiv (C_i \wedge C_j \wedge \mathit{false})$$

$$\equiv \mathit{false}$$

That is we can remove $(C_i \wedge C_j \wedge \neg(A_i = A_j))$ if $A_i = A_j$ is true since $\mathit{false} \vee X \equiv X$. If $A_i \neq A_j$, we have:

$$(C_i \wedge C_j \wedge \neg(A_i = A_j))$$

$$\equiv (C_i \wedge C_j \wedge \mathit{true})$$

$$\equiv (C_i \wedge C_j)$$

Thus, the sentence for checking contradiction has the same form as the sentence for checking redundancy except where $A_i = A_j$. The negated testing sentence can be converted into clausal form. The converted clauses can be tested for contradiction using the resolution principle.

Let us now consider the same set of rules for the Method Object "cb" used in the previous sections. The resulting clauses converted from the negated testing sentence by the procedure described in this section are:

$\{ U, G \}$ (a)
$\{ \neg U, G \}$ (b)
$\{ \neg U \}$ (c)
$\{ \neg G \}$ (d)
$\{ \neg U, \neg G \}$. (e)

By performing resolution on these clauses, we obtain

$\{ G \}$ from (a) and (c) (f)
$\{ \}$ from (d) and (f) (g)

which is an empty clause. Thus, the rules are not in contradiction. Although the first and second rules are judged as redundant, they do not contradict because the two redundant rules have the same action (conclusion) parts. Therefore, these rules do not have a contradiction problem. The session of checking lack of contradiction of the Method Object "cb" using HyperLRFD++ is shown in Figure 7-6.

7.1.4 Compressing Multiple Method Objects

In the preceding sections, syntactic checking of an individual Method Object that does not depend on other Method Objects is described. A rule in a Method Object may contain other Method Objects in its condition part. Since the logic of the rule depends on the logic of these referenced Method Objects, the referenced Method Objects must be incorporated in order to check the syntactic properties of the Method Object. This section describes a procedure for checking Method Objects that depend on other Method Objects.

If the Standards Analysis Module finds a referenced Method Object in the condition part, it parses the rules of the referenced Method Object and substitutes the condition with the conditions of the referenced Method Object. This procedure is recursively called until all the conditions are found to be independent of other Method Objects. Then, the module removes conditions which are not directly related to logic of the provision, such as "self<-respond_satisfied" and "self<-input_attr(Mem, load_comp, 1000)." Once the condition list is made, the checking procedure of completeness and uniqueness is the same as described in the preceding sections.

```
:-  contradiction

ConditionList =
[[_1133::unbraced_cantilever_y=yes], [_1133::greater_moment_x=yes],
 [not (_1133::unbraced_cantilever_y=yes),
  not (_1133::greater_moment_x=yes)]]
ActionList =
[cb(_1133,1,_1153), cb(_1133,1,_1153), cb(_1133,_1167,_1153)]

NegatedTestingSentence =
_1606::unbraced_cantilever_y=yes&not (_1606::unbraced_cantilever_y=yes)&
not (_1606::greater_moment_x=yes)v _1606::greater_moment_x=yes&not
(_1606::unbraced_cantilever_y=yes)&not (_1606::greater_moment_x=yes)v
_1606::greater_moment_x=yes&not (_1606::unbraced_cantilever_y=yes)&
not (_1606::greater_moment_x=yes)

CLAUSES
  *** clause2(_1606::greater_moment_x=yes v
             _1606::unbraced_cantilever_y=yes)
  *** clause2(_1606::greater_moment_x=yes v
             not (_1606::unbraced_cantilever_y=yes))
  *** clause2(not (_1606::unbraced_cantilever_y=yes))
  *** clause2(not (_1606::greater_moment_x=yes))
  *** clause2(not (_1606::greater_moment_x=yes)v
             _1606::unbraced_cantilever_y=yes)
  *** clause2(not (_1606::greater_moment_x=yes)v
             not (_1606::unbraced_cantilever_y=yes))

PERFORMING RESOLUTION
_3308::greater_moment_x=yes
[] : Empty clause found
GOOD! The given rules do NOT CONTRADICT.
Nº1        yes
```

Figure 7-6 The Session of Checking Lack of Contradiction

Let us consider an example of a requirement Method Object “req_short_non_slender_prism_y_comp_mem” shown in Figure 7-7. The condition list for this Method Object is

```
[[Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
  DS is 0.85 * Pn,
  self <- input_attr(Mem,load_comp,1000),
  self <- eqless(Mem::load_comp,DS),
  self <- respond_satisfied],
 [Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
  DS is 0.85 * Pn,
  not(self <- eqless(Mem::load_comp,DS)),
  self <- respond_violated]].
```

```
open_object req_short_non_slender_prism_y_comp_mem.

  super = requirements.
  class = short_non_slender_prism_y_comp_mem.
  provision = 'E2'.
  reference = [pn_short_y,
               attr(load_comp)].
  meaning = 'requirement for a non-slender prismatic compression
             member which y-axis inelastic buckling governs'.

  req_short_non_slender_prism_y_comp_mem(Mem,satisfied,Id):-
      Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
      DS is 0.85 * Pn,
      self <- input_attr(Mem,load_comp,1000),
      self <- eqless(Mem::load_comp,DS),
      self <- respond_satisfied.

  req_short_non_slender_prism_y_comp_mem(Mem,violated,Id):-
      Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
      DS is 0.85 * Pn,
      not(self <- eqless(Mem::load_comp,DS)),
      self <- respond_violated.

close_object req_short_non_slender_prism_y_comp_mem.
```

Figure 7-7   An Example of Requirement Method Objects

Since "pn_short_y" in the first item of the list is a referenced Method Object, the module substitutes the first item in the condition list with the conditions of the rules in the Method Object "pn_short_y." The condition list now becomes

```
[[Mem::gross_area <- gross_area(Mem,Ag,Id),
  Mem::fcry_short<-fcry_short(Mem,Fcry,Id),
  Pn is Ag * Fcry,
  DS is 0.85 * Pn,
  self <- input_attr(Mem,load_comp,1000),
  self <- eqless(Mem::load_comp,DS),
  self <- respond_satisfied],
 [Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
  DS is 0.85 * Pn,
  not(self <- eqless(Mem::load_comp,DS)),
  self <- respond_violated]].
```

Again, the first item of the condition list has a referenced Method Object "gross_area." Thus, the module substitutes it with the conditions of the rule in the Method Object "gross_area." The condition list is transformed as

```
[[Mem<-area(Id,Ag),
  Mem::fcry_short<-fcry_short(Mem,Fcry,Id),
  Pn is Ag * Fcry,
  DS is 0.85 * Pn,
  self <- input_attr(Mem,load_comp,1000),
  self <- eqless(Mem::load_comp,DS),
  self <- respond_satisfied],
 [Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
  DS is 0.85 * Pn,
  not(self <- eqless(Mem::load_comp,DS)),
  self <- respond_violated]].
```

By iterating this procedure, the system finally obtains the condition list as:

```
[[Mem<-area(Id,Ag),
  Mem<-ry(Id,Ry),
  Lambda_c_y is Mem::effective_length_y * Mem::unbraced_length_y /
      pi / Ry * sqrt(Mem::yield_stress / Mem::elastic_modulus),
  Fy = Mem::yield_stress,
  Fcry is 0.658 ^ (Lambda_c_y ^ 2) * Fy,
  Pn is Ag * Fcry,
  DS is 0.85 * Pn,
  self <- input_attr(Mem,load_comp,1000),
  self <- eqless(Mem::load_comp,DS),
  self <- respond_satisfied],
 [Mem<-area(Id,Ag),
  Mem<-ry(Id,Ry),
  Lambda_c_y is Mem::effective_length_y * Mem::unbraced_length_y /
      pi / Ry * sqrt(Mem::yield_stress / Mem::elastic_modulus),
  Fy = Mem::yield_stress,
  Fcry is 0.658 ^ (Lambda_c_y ^ 2) * Fy,
  Pn is Ag * Fcry,
  DS is 0.85 * Pn,
  not(self <- eqless(Mem::load_comp,DS)),
  self <- respond_violated]].
```

By removing unrelated condition items, the final condition list is obtained as follows:

```
[[self <- eqless(Mem::load_comp,DS)],
 [not(self <- eqless(Mem::load_comp,DS))]].
```

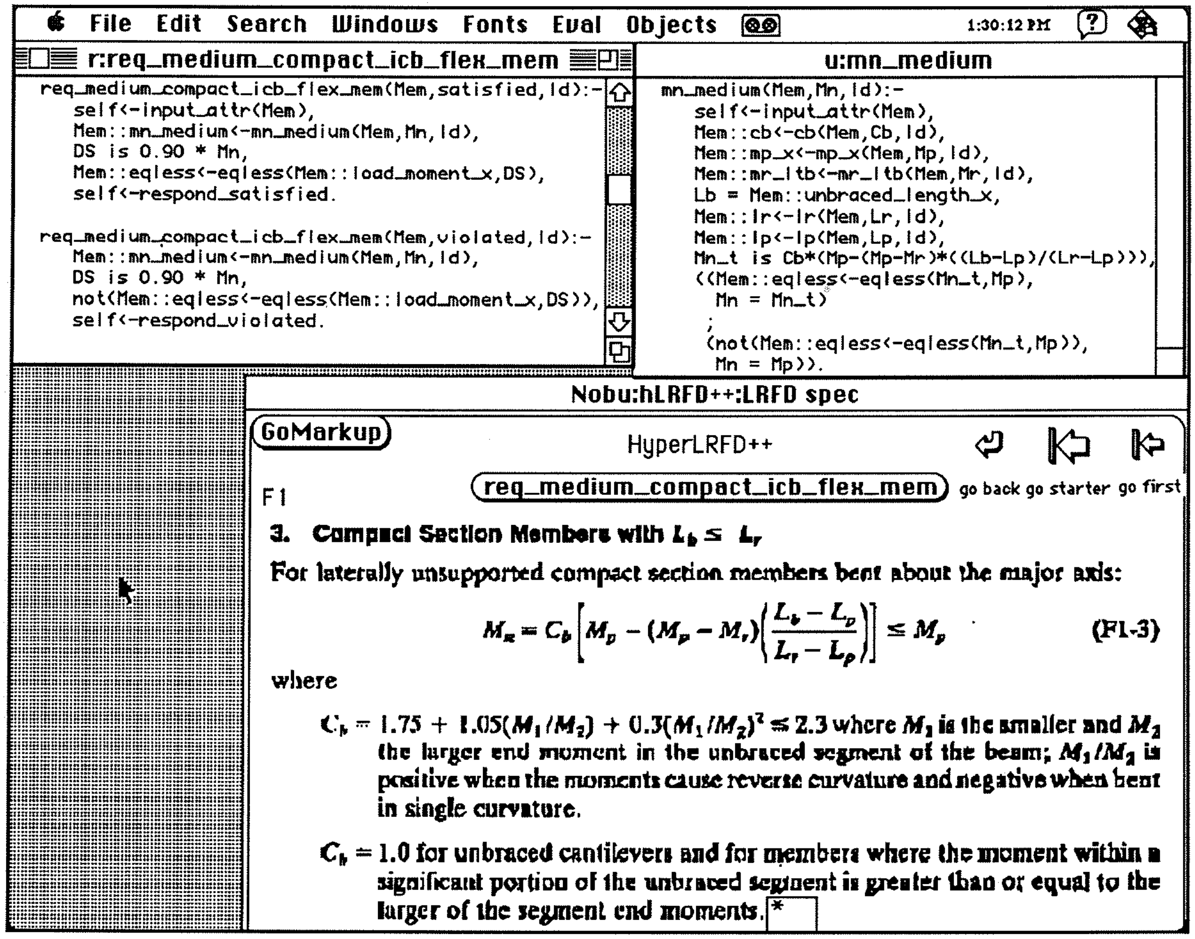
The action list is

```
[req_short_non_slender_prism_y_comp_mem(Mem,satisfied,Id),
 req_short_non_slender_prism_y_comp_mem(Mem,violated,Id)].
```

These condition and action lists are then used to check completeness, lack of redundancy, and lack of contradiction as described in the preceding sections.

7.1.5 Semantic Checking of Individual Provisions

As noted in Chapter 1, current standards processing models lack of mappings between design provisions and program code representing the provisions. Thus, it is difficult to check whether the program code correctly represents the corresponding provision. As described in Chapter 5, both provisions and program code are represented as HyperDocument and have pointers to each other. Thus, the user can read the program code (the Method Object) of a specific provision by triggering the pointers to the Method Objects representing that provision. Both the provision and the program code then appear on the terminal screen simultaneously as shown in Figure 7-8. This capability provides the programmer or the user a means to check the program code against the semantic content of the provision as to whether the intention of the provision is correctly reflected in Object-Logic sentences.



```
req_medium_compact_icb_flex_mem(Mem,satisfied,Id):-
   self<-input_attr(Mem),
   Mem::mn_medium<-mn_medium(Mem,Mn,Id),
   DS is 0.90 * Mn,
   Mem::eqless<-eqless(Mem::load_moment_x,DS),
   self<-respond_satisfied.

req_medium_compact_icb_flex_mem(Mem,violated,Id):-
   Mem::mn_medium<-mn_medium(Mem,Mn,Id),
   DS is 0.90 * Mn,
   not(Mem::eqless<-eqless(Mem::load_moment_x,DS)),
   self<-respond_violated.
```

```
mn_medium(Mem,Mn,Id):-
   self<-input_attr(Mem),
   Mem::cb<-cb(Mem,Cb,Id),
   Mem::mp_x<-mp_x(Mem,Mp,Id),
   Mem::mr_ltb<-mr_ltb(Mem,Mr,Id),
   Lb = Mem::unbraced_length_x,
   Mem::lr<-lr(Mem,Lr,Id),
   Mem::lp<-lp(Mem,Lp,Id),
   Mn_t is Cb*(Mp-(Mp-Mr)*((Lb-Lp)/(Lr-Lp))),
   ((Mem::eqless<-eqless(Mn_t,Mp),
     Mn = Mn_t)
    ;
    (not(Mem::eqless<-eqless(Mn_t,Mp)),
     Mn = Mp)).
```

3. Compact Section Members with $L_b \leq L_r$

For laterally unsupported compact section members bent about the major axis:

$$M_n = C_b\left[M_p - (M_p - M_r)\left(\frac{L_b - L_p}{L_r - L_p}\right)\right] \leq M_p \qquad \text{(F1-3)}$$

where

C_b = 1.75 + 1.05(M_1/M_2) + 0.3(M_1/M_2)^2 ≤ 2.3 where M_1 is the smaller and M_2 the larger end moment in the unbraced segment of the beam; M_1/M_2 is positive when the moments cause reverse curvature and negative when bent in single curvature.

C_b = 1.0 for unbraced cantilevers and for members where the moment within a significant portion of the unbraced segment is greater than or equal to the larger of the segment end moments. *

Figure 7-8 A Computer Screen Showing Method Objects and a Provision

7.2 Analysis of the Standards Organization

The Standards Analysis Module also examines whether the standard is appropriately organized by checking the completeness, uniqueness, and correctness of sentences of each classification Method Object and by showing the Member Class Hierarchy to the user. Each classification Method Object is associated with its corresponding OR node in the Member Class Hierarchy. At the OR node of the Member Class Hierarchy, the system classifies a design member into a single specific child node class by executing the classification method. Thus, the rules of the classification Method Object must cover all the possible conditions and a unique rule must be applicable for any condition. Completeness of the classification Method Objects ensures that the organization of the standard is collectively exhaustive at every OR node level, and uniqueness ensures that the organization is mutually exclusive at every OR node to guarantee that each requirement is uniquely described by a single path in the Member Class Hierarchy. Correctness ensures that the provision is correctly represented as the rules in the classification Method Object. The method of checking the completeness and uniqueness of classification Method Objects is the same as the one described in the previous section.

In this section, the procedures for checking these properties of the classification Method Object are described by checking a sample classification Method Object. Suppose that a OR node class "steel_mem," denoting steel member, is divided into three subclasses "comp_mem," "flex_mem," and "flex_comp_mem," denoting compression member, flexural member, and flexural compression member, respectively, according to the two limited stress states, i.e., compression and flexure by the simplified classification Method Object "clas_steel_mem" as shown in Figure 7-9. In the following subsections, this sample classification Method Object is used.

```
classify(Mem,comp_mem,ID):-
      self<-greater(Mem::load_comp,0),
      not(self<-greater(Mem::load_moment,0)).
classify(Mem,flex_mem,ID):-
      self<-greater(Mem::load_moment,0),
      not(self<-greater(Mem::load_comp,0)).
classify(Mem,flex_comp_mem,ID):-
      self<-greater(Mem::load_comp,0),
      self<-greater(Mem::load_moment,0).
```

where `greater(A,B):  A > B,`

`load_comp`: required compressive strength based on the factored nominal loads,

`load_moment`: required flexural strength based on the factored nominal loads.

Note: both `load_comp` and `load_moment` attributes reject values less than zero.

Figure 7-9 Rules in the Method Object "clas_steel_mem"

7.2.1 Checking Completeness of the Classification Method Object

To check the completeness of a set of classification rules in the Method Object, the system first parses the rules and obtains the condition and action lists. The lists for the classification Method Object "clas_steel_mem" are:

```
ConditionList =
      [[self<-greater(Mem::load_comp,0),
        not(self<-greater(Mem::load_moment,0))],
       [self<-greater(Mem::load_moment,0),
        not(self<-greater(Mem::load_comp,0))],
       [self<-greater(Mem::load_comp,0),
        self<-greater(Mem::load_moment,0)]].

ActionList = [classify(Mem,comp_mem,ID),
              classify(Mem,flex_mem,ID),
              classify(Mem,flex_comp_mem,ID)].
```

The system automatically negates the sentence for checking completeness corresponding to the condition list and converts it into clausal form:

```
{not self<-greater(Mem::load_comp,0),
 self<-greater(Mem::load_moment,0)}
{not self<-greater(Mem::load_moment,0),
 self<-greater(Mem::load_comp,0)}
```

```
{not self<-greater(Mem::load_moment,0),
 not self<-greater(Mem::load_comp,0)}
```

By performing resolution on these clauses, we obtain:

```
{not self<-greater(Mem::load_comp,0),
 self<-greater(Mem::load_comp,0)}
{not self<-greater(Mem::load_comp,0)}
{not self<-greater(Mem::load_moment,0)}
{not self<-greater(Mem::load_moment,0),
 self<-greater(Mem::load_moment,0)
{not self<-greater(Mem::load_moment,0),
 not self<-greater(Mem::load_comp,0)}
```

Since the empty clause cannot be obtained from the above clauses, the rule set is not complete. The reason that the rules are not complete is that if the member's stress state is not compression nor flexure, no rules are applicable in that rule set. A rule with the condition:

```
not self<-greater(Mem::load_moment,0),
not self<-greater(Mem::load_comp,0)
```

which would classify the member as a tension member or a flexural compression member is necessary. The session of checking completeness by using HyperLRFD++ is shown in Figure 7-10.

7.2.2 Checking Lack of Redundancy of the Classification Method Object

To check the lack of redundancy using HyperLRFD++, the system generates the negated testing sentence from the same condition list generated in the previous section. The system automatically converts the testing sentence into clausal form:

```
{self<-greater(Mem::load_comp,0),
 self<-greater(Mem::load_moment,0)}
{self<-greater(Mem::load_comp,0)}
{self<-greater(Mem::load_comp,0),
 not self<-greater(Mem::load_moment,0)}
{not self<-greater(Mem::load_moment,0)}
{self<-greater(Mem::load_moment,0)}
```

```
:-  completeness

ConditionList =
[[self<-greater(_1273::load_comp, 0),
  not (self<-greater(_1273::load_moment, 0))],
 [self<-greater(_1273::load_moment, 0),
  not (self<-greater(_1273::load_comp, 0))],
 [self<-greater(_1273::load_comp, 0),
  self<-greater(_1273::load_moment, 0)]]

NegatedTestingSentence = not (self<-greater(_1841::load_comp, 0)&
not (self<-greater(_1841::load_moment, 0))v
self<-greater(_1841::load_moment, 0)&
not (self<-greater(_1841::load_comp, 0))v
self<-greater(_1841::load_comp, 0)&self<-greater(_1841::load_moment, 0))

CLAUSES
  *** clause2(not (self<-greater(_1841::load_comp, 0))v
              self<-greater(_1841::load_moment, 0))
  *** clause2(not (self<-greater(_1841::load_moment, 0))v
              self<-greater(_1841::load_comp, 0))
  *** clause2(not (self<-greater(_1841::load_comp, 0))v
              not (self<-greater(_1841::load_moment, 0)))

PERFORMING RESOLUTION
not (self<-greater(_2788::load_comp, 0))v
      self<-greater(_2788::load_comp, 0)
not (self<-greater(_3165::load_comp, 0))v
      not (self<-greater(_3165::load_comp, 0))
not (self<-greater(_4110::load_moment, 0))
not (self<-greater(_4455::load_moment, 0))v
      self<-greater(_4455::load_moment, 0)
not (self<-greater(_4832::load_moment, 0))v
      not (self<-greater(_4832::load_moment, 0))
not (self<-greater(_6020::load_moment, 0))v
      not (self<-greater(_6020::load_comp, 0))
not (self<-greater(_6406::load_comp, 0))v
      not (self<-greater(_6406::load_comp, 0))
not (self<-greater(_7744::load_moment, 0))v
      not (self<-greater(_7744::load_moment, 0))
Empty clause NOT found
NOT GOOD!  The given rules are NOT COMPLETE.
Nº1         yes
```

Figure 7-10 The Session of Checking Completeness

```
     {self<-greater(Mem::load_moment,0),
      not self<-greater(Mem::load_comp,0)}
```

By performing resolution on these clauses, we obtain an empty clause. Thus, the given rules are not redundant. Figure 7-11 shows the session of this process.

```
:-  redundancy

ConditionList =
[[self<-greater(_1003::load_comp, 0),
  not (self<-greater(_1003::load_moment, 0))],
 [self<-greater(_1003::load_moment, 0),
  not (self<-greater(_1003::load_comp, 0))],
 [self<-greater(_1003::load_comp, 0),
  self<-greater(_1003::load_moment, 0)]]

NegatedTestingSentence = (self<-greater(_1810::load_comp, 0)&
not (self<-greater(_1810::load_moment, 0)))&
self<-greater(_1810::load_moment, 0)&
not (self<-greater(_1810::load_comp, 0))v
(self<-greater(_1810::load_comp, 0)&
not (self<-greater(_1810::load_moment, 0)))&
self<-greater(_1810::load_comp, 0)&
self<-greater(_1810::load_moment, 0)v
(self<-greater(_1810::load_moment, 0)&
not (self<-greater(_1810::load_comp, 0)))&
self<-greater(_1810::load_comp, 0)&
self<-greater(_1810::load_moment, 0)v
(self<-greater(_1810::load_moment, 0)&
not (self<-greater(_1810::load_comp, 0)))&
self<-greater(_1810::load_comp, 0)&
self<-greater(_1810::load_moment, 0)

CLAUSES
  *** clause2(self<-greater(_1810::load_comp, 0)v
              self<-greater(_1810::load_moment, 0))
  *** clause2(self<-greater(_1810::load_comp, 0))
  *** clause2(self<-greater(_1810::load_comp, 0)v
              not (self<-greater(_1810::load_moment, 0)))
  *** clause2(not (self<-greater(_1810::load_moment, 0)))
  *** clause2(self<-greater(_1810::load_moment, 0))
  *** clause2(self<-greater(_1810::load_moment, 0)v
              not (self<-greater(_1810::load_comp, 0)))

PERFORMING RESOLUTION
[] : Empty clause found
GOOD! The given rules are NOT REDUNDANT.
Nº1          yes
```

Figure 7-11 The Session of Checking Lack of Redundancy

7.2.3 Checking Lack of Contradiction of the Classification Method

To check the lack of contradiction using HyperLRFD++, the system generates the negated testing sentence from the same condition and action lists generated earlier. In this case, since the items in the action list are mutually different, the result of checking the lack of redundancy is the same as checking the lack of contradiction. Figure 7-12 shows the session of this process. Since the empty clause is found in this case, the module declares that the given rules do not contradict.

7.2.4 Checking Correctness of the Standards Organization

As described in Section 7.1.5, semantic correctness of the Method Object is checked by comparing the program code and the corresponding provision. As for the classification Method Object, more than one provision may correspond to the classification method. For example, the classification Method Object "clas_steel_mem" corresponds to the three provisions as shown in Figure 7-13, which can be found in AISC LRFD specification [AISC 86]. By comparing these provisions and the rules of the classification Method Object, the user can check the correctness of Method Object.

A Member Class Hierarchy that is represented graphically, as shown in Figure 7-14, can serve as a means to check whether the organization is correctly implemented in the design software. Prolog++ provides a graphical user interface, where the user can develop a program by creating a class hierarchy on a computer screen (Figure 7-14). This feature is used in the implementation of HyperLRFD++.

```
:-  contradiction

ConditionList =
[[self<-greater(_1003::load_comp, 0),
  not (self<-greater(_1003::load_moment, 0))],
 [self<-greater(_1003::load_moment, 0),
  not (self<-greater(_1003::load_comp, 0))],
 [self<-greater(_1003::load_comp, 0),
  self<-greater(_1003::load_moment, 0)]]
ActionList =
[classify(_1003,comp_mem,_1267), classify(_1003,flex_mem,_1267),
 classify(_1003,flex_comp_mem,_1267)]

NegatedTestingSentence = (self<-greater(_1845::load_comp, 0)&
not (self<-greater(_1845::load_moment, 0)))&
self<-greater(_1845::load_moment, 0)&
not (self<-greater(_1845::load_comp, 0))v
(self<-greater(_1845::load_comp, 0)&
not (self<-greater(_1845::load_moment, 0)))&
self<-greater(_1845::load_comp, 0)&
self<-greater(_1845::load_moment, 0)v
(self<-greater(_1845::load_moment, 0)&
not (self<-greater(_1845::load_comp, 0)))&
self<-greater(_1845::load_comp, 0)&
self<-greater(_1845::load_moment, 0)v
(self<-greater(_1845::load_moment, 0)&
not (self<-greater(_1845::load_comp, 0)))&
self<-greater(_1845::load_comp, 0)&
self<-greater(_1845::load_moment, 0)

CLAUSES
   *** clause2(self<-greater(_1845::load_comp, 0)v
               self<-greater(_1845::load_moment, 0))
   *** clause2(self<-greater(_1845::load_comp, 0))
   *** clause2(self<-greater(_1845::load_comp, 0)v
               not (self<-greater(_1845::load_moment, 0)))
   *** clause2(not (self<-greater(_1845::load_moment, 0)))
   *** clause2(self<-greater(_1845::load_moment, 0))
   *** clause2(self<-greater(_1845::load_moment, 0)v
               not (self<-greater(_1845::load_comp, 0)))

PERFORMING RESOLUTION
[] : Empty clause found
GOOD! The given rules do NOT CONTRADICT.
Nº1        yes
```

Figure 7-12 The Session of Checking the Lack of Contradiction

CHAPTER E. COLUMNS AND OTHER COMPRESSION MEMBERS
This section applies to prismatic members subject to axial compression through the centroidal axis. For members subject to combined axial compression and flexure, see Chap. H. For tapered members, see Appendix F4.

CHAPTER F. BEAMS AND OTHER FLEXURAL MEMBERS
This section applies to singly or doubly symmetric beams including hybrid beams and girders loaded in the plane of symmetry. It also applies to channels loaded in a plane passing through the shear center parallel to the web or restrained against twisting at the load points and points of support. For design flexural strength for members not covered in Sect. F1, see Appendix F1.7. For members subject to combined flexural and axial force, see Sect. H1. For unsymmetric beams and beams subject to torsion combined with flexure, see Sect H2.

CHAPTER H. MEMBERS UNDER TORSION AND COMBINED FORCES
This section applies to prismatic members subjected to axial force and flexure about one or both axes of symmetry, with or without torsion, and torsion only. For web-tapered members, see Appendix F4.

Figure 7-13 Provisions Corresponding to the Classification Method "clas_steel_mem" [AISC 86]

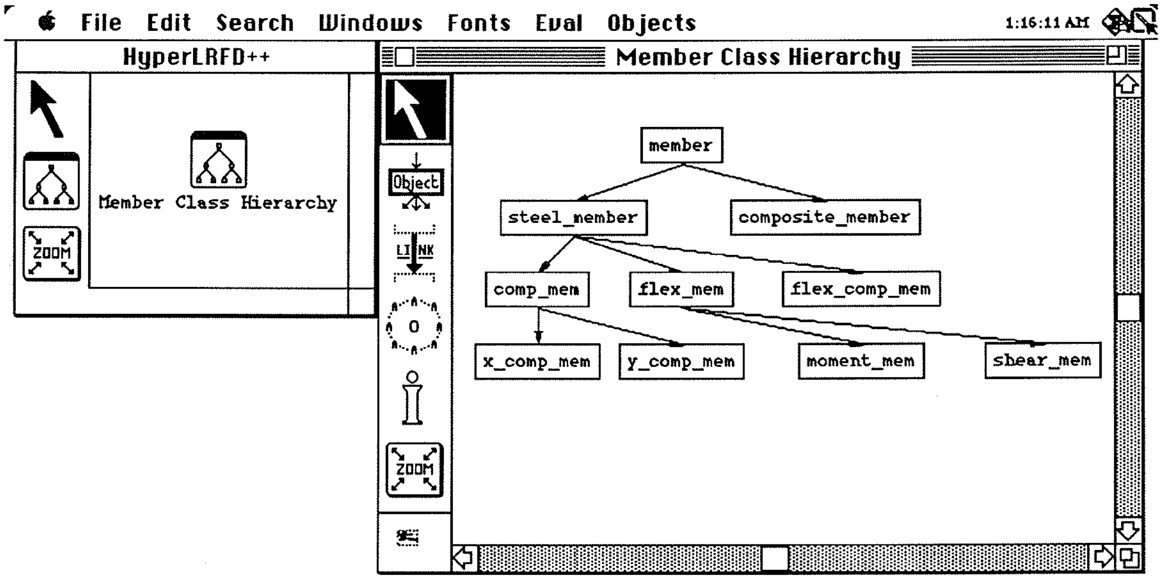


Figure 7-14 Graphical User Interface to Create a Member Class Hierarchy

7.3 Checking Relations Among Provisions

The Standards Analysis Module checks whether provisions are connected and acyclic by examining the dependency network of Method Objects based on their relationships. Each Method Object has a <reference> attribute. The <reference> attribute value is obtained by parsing the Object-Logic sentences of the method and collecting the Method Objects' names in the condition part. For example, the <reference> attribute value of the Method Object "req_short_non_slender_prism_y_comp_mem" shown in Figure 7-15 is:

```
        [pn_short_y, attr(load_comp)].
```

The procedure to generate a <reference> attribute value is to identify "Mem::" in the condition part, then determine if the term after "Mem::" is a Method Object or an attribute based on the existence of "<-" after the term (if "<-" exists, it is a Method Object). If it is an attribute, "attr( )" is attached to the term. Note that the module ignores unrelated conditions such as "eqless," "input_attr," and "respond_satisfied" so that they are not included in the <reference> attribute. For each requirement a dependency network can be drawn by calling <reference> attributes recursively. For example, dependency network of the requirement Method Object "req_short_non_slender_prism_y_comp_mem" is shown in Figure 7-15.

The module checks the following two cases for connectedness:

- A Method Object has a non-existing referencing Method Object. For example, if the Method Object "pn_short_y" in Figure 7-16 does not exist, the network is disconnected.
- A leaf node of the dependency network is not a basic data item, i.e., an attribute of design member objects. This situation occurs when the leaf node Method Object of the dependency network has no conditions in the rules. All the leaf nodes of the dependency network in Figure 7-16 are attributes of design member objects.

The module can also generate another dependency network replacing the Method Object name with its meaning, which is stored in the <meaning> attribute of each Method Object. For example, the dependency network shown in Figure 7-17 can be converted as shown in Figure 7-17. This network can be used to check semantic correctness of the Method Objects dependency.

```
open_object req_short_non_slender_prism_y_comp_mem.

  super = requirements.
  class = short_non_slender_prism_y_comp_mem.
  provision = 'E2'.
  reference = [pn_short_y,
               attr(load_comp)].
  meaning = 'requirement for a non-slender prismatic compression
             member which y-axis inelastic buckling governs'.

  req_short_non_slender_prism_y_comp_mem(Mem,satisfied,Id):-
      Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
      DS is 0.85 * Pn,
      self <- input_attr(Mem,load_comp,1000),
      self <- eqless(Mem::load_comp,DS),
      self <- respond_satisfied.

  req_short_non_slender_prism_y_comp_mem(Mem,violated,Id):-
      Mem::pn_short_y <- pn_short_y(Mem,Pn,Id),
      DS is 0.85 * Pn,
      not(self <- eqless(Mem::load_comp,DS)),
      self <- respond_violated.

close_object req_short_non_slender_prism_y_comp_mem.
```

Figure 7-15 An Example of Requirement Method Objects

```
req_short_non_slender_prism_y_comp_mem
   |- pn_short_y
   |     |- gross_area
   |     |     |- attr(area)
   |     |- fcry_short
   |           |- lambda_c_y
   |           |     |- radius_gyration_y
   |           |     |     |- attr(ry)
   |           |     |- attr(effective_length_factor_y)
   |           |     |- attr(unbraced_length_y)
   |           |     |- attr(yield_stress)
   |           |     |- attr(elastic_modulus)
   |           |- attr(yield_stress)
   |- attr(load_comp)
```

Figure 7-16 Dependency Network of the Requirement Method Object

```
requirement for a non-slender prismatic compression member which y-axis
inelastic buckling governs
   |
   |- nominal compressive strength for a compression member which y-axis
   |  inelastic buckling governs
   |        |
   |        |- gross area of a section
   |        |      |- attr(area)
   |        |
   |        |- critical stress of a compression member which y-axis
   |           inelastic buckling governs
   |                |
   |                |- column slenderness parameter for the y-axis
   |                |
   |                |- radius of gyration of the y-axis
   |                |      |- attr(ry)
   |                |
   |                |- attr(effective_length_factor_y)
   |                |      |- attr(effective_length_factor_y)
   |                |      |- attr(unbraced_length_y)
   |                |      |- attr(yield_stress)
   |                |      |- attr(elastic_modulus)
   |                |- attr(yield_stress)
   |- attr(load_comp)
```

Figure 7-17 Dependency Network of the Requirement Method Object with Meaning

If a network contains a branch that the same Method Object appears more than twice in one path, the network is cyclic. For example, if in case the Method Object "fcry_short" happens to have a referencing Method Object "pn_short_y," the network is cyclic as shown in Figure 7-18. If the network has a cyclic branch, the system warns the user.

7.4 Summary

This chapter described the methodologies for standards analysis, employed in the Hyper-Object-Logic model. First, the checking of individual provisions was described. Then, the checking of standards organization was discussed. Finally, checking of relationships among provisions was described. Illustrative examples were shown for each checking task by using the prototype system, HyperLRFD++.

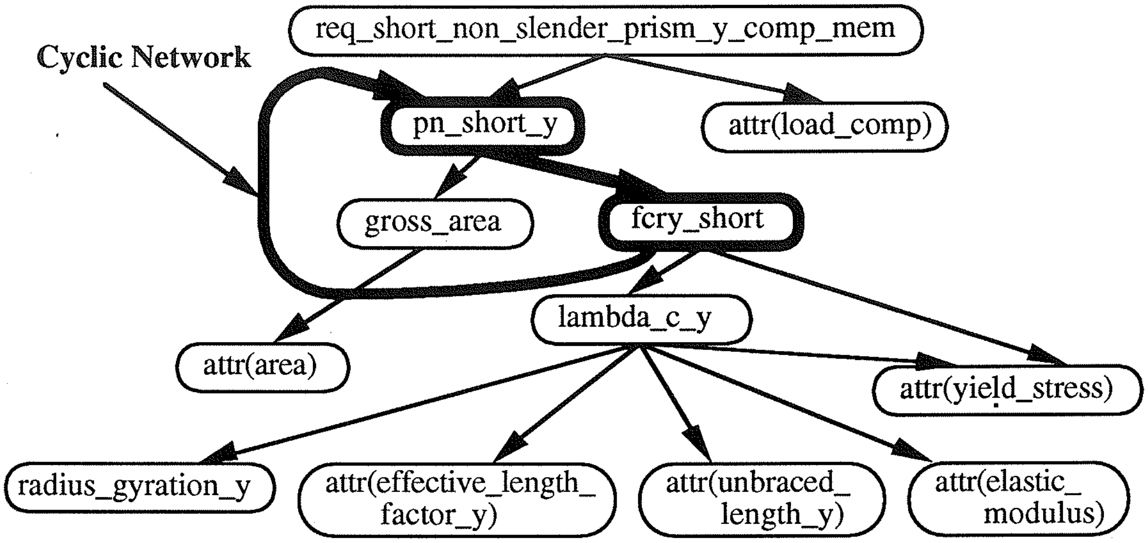


Figure 7-18 A Cyclic Network (A Hypothetical Example)

Chapter 8
Summary and Discussion

The objective of this thesis is to develop a model that can perform both conformance checking and component design, store background information of design standards, and analyze design codes in an intelligent CAD environment. This thesis proposed and demonstrated an integrated Hyper-Object-Logic model for the representation, processing, and documentation of design standards. This chapter provides a brief summary and discusses contributions of this thesis. Furthermore, directions for future work are discussed.

8.1 Summary and Contributions

This thesis proposed and demonstrated an Object-Logic model for design standards processing. This model combines object-oriented and logic programming paradigms to provide a unified framework for the representation and processing of the design standard. This thesis has shown that the object-oriented paradigm is suitable for representing the organization and data items of the design standard. In addition, the logic programming paradigm is well-suited for representing and processing design provisions both for design and conformance checking, as well as checking syntactic completeness and uniqueness of the design standard. In addition to the Object-Logic model, this thesis described and demonstrated a HyperDocument model, which is based on the current development of HyperFile. This model serves as a large heterogeneous document storage and manipulation system useful for code developers and design engineers. The Object-Logic and the HyperDocument models were integrated into a unified Hyper-Object-Logic model.

For more than two decades, researchers and engineers have attempted to automate design standards processing. In this thesis, the SASE model and several recently proposed Artificial Intelligence-based approaches were examined. The Object-Logic model is partially based on the previous development of the SASE model [Fenves 87], the frame-based and object-oriented representation of design standards [Garrett 86] [Garrett 89], and the logic-based approach [Chan 86] [Jain 89].

In the Object-Logic model, the organization of the standard is represented as an object-oriented AND-OR class hierarchy (Member Class Hierarchy) and design provisions are represented declaratively as Object-Logic sentences in the Method Objects. Since the provisions are separately implemented from the inference mechanism of logic programming, future revision of the provisions will require modifying the corresponding Method Objects only.

Once the organization and provisions of the design standard are implemented in the Object-Logic model, the user can perform both conformance checking and component design within the same design environment. Conformance checking involves identifying and executing all the applicable requirements of a given design member object. The system identifies all the applicable requirements by traversing the Member Class Hierarchy, based on the attribute values of the design member object. The system executes the applicable requirements by the resolution principle of logic and message passing among Method Objects. To automate component design using design standards, a heuristics-based generate-and-test method was used. To define a design member, an Object Model, which facilitates creating an object and entering attribute values, was defined. To retrieve data not included in the standard, such as dimensions and properties of structural standard shapes, a relational database system has been integrated with the system. Since relational calculus is a subset of predicate calculus, a seamless integration between the Object-Logic model and the relational database system can be achieved. This model could also be integrated with other design application programs such as drafting systems, analysis programs, and system design programs, by sharing design member object data through appropriate interfaces.

The HyperDocument model consists of a Document Base and a Navigation system. Each document is composed of the content, which is a file, and its HyperTag, which contains information about the document and pointers to the referenced and related documents. The idea is to provide a generic heterogeneous document storage and manipulation system. When developing a design standard, the background information and data about the provisions can be stored in the system and retrieved when necessary. Furthermore, the background information stored in the system can be used to help engineers understand implications of the provisions. This information is especially helpful for the re-design purpose when a design member object violates a requirement. One feature of the HyperDocument model is the inclusion of external programs to compute design data that require processing of charts, tables, or complex equations.

The Hyper-Object-Logic model includes a facility to analyze design standards. The system can check completeness, uniqueness, and correctness at the individual provision and at the organizational levels. The method for syntactic checking of completeness and uniqueness is based on the procedure proposed by Jain et al. [Jain 89]. Semantic correctness checking involves comparing design provision documents and the corresponding program codes by the mappings between these documents in the HyperDocument model. The system can also check whether the dependency network of provisions is acyclic, connected, and semantically correct. Checking these properties can ensure correctness of both the design standard and the design program representing the standard.

To demonstrate the feasibility and practicality of the Hyper-Object-Logic model, a prototype system, HyperLRFD++, was implemented for a part of the AISC LRFD specification [AISC 86]. A number of examples have been provided to demonstrate the capabilities of the Hyper-Object-Logic model described in this thesis.

The results of this thesis can be summarized as follows:

- A new representation scheme for design standards processing through a combination of object-oriented and logic programming paradigms was developed.
- A methodology to perform both conformance checking and component design within the same design environment was established using the Object-Logic scheme.

- A model for storing and utilizing heterogeneous documents of background information and knowledge was developed and integrated with the design standards processing model.
- A framework for checking completeness, uniqueness, and correctness of design standards was developed.
- A framework for integrating a design standards processing system with other design applications was developed.

The contributions of this thesis can be summarized as follows:

- Demonstrates the combination of the object-oriented and logic programming paradigms is suitable for representing the organization and data items of the design standard and for representing and processing design provisions both for component design and conformance checking.
- Demonstrates the capabilities of the hypertext and HyperFile technologies for storing and utilizing heterogeneous documents of design standards and their background information and knowledge.
- Introduces and demonstrates the importance of storing and utilizing background information both for code developers and design engineers.
- Demonstrates the importance and effects of integrating the model for the representation and processing of design standards with the model for the documentation of design standards and their background information and knowledge.
- Demonstrates and implements the framework for checking completeness, uniqueness, and correctness of design standards.
- Introduces the potential for integrating the design standards processing system with other design applications such as analysis, system design, and drawing.

In the next section, limitations of this research and directions for future work are discussed.

8.2 Limitations of the Research and Future Work

Since HyperLRFD++ has been developed as a demonstration system for the feasibility study of the Hyper-Object-Logic model, it contains limitations in several areas. In this section, limitations of the prototype system and the model are discussed and directions for future work are introduced.

One of the limitations is partial coverage of the AISC LRFD specification, i.e., it includes the member types of compression, flexural, and flexural-compression steel members. One extension will include the design provisions concerning tension members and connections in HyperLRFD++. For these types of components, a graphical user interface representing the layout of bolts or welds should be incorporated. Other external programs such as calculating the effective net area for tension members should also be included. The object-oriented paradigm can play an effective role in dealing with the interactions between the connections and the connected member objects.

Second limitation is that HyperLRFD++ deals with only standard wide flange W shape sections. Thus, other standard shape sections and arbitrary shapes such as built-up members should also be implemented. Although checking an arbitrary shape member is not a difficult task, generating an arbitrary shape will require high level heuristic knowledge, which may not be found in books or papers but are based on engineers' experience. From the structural engineering point of view, however, a systematic methodology for design needs to be addressed instead of merely implementing an engineer's ad-hoc procedures for generating an arbitrary shape. The design methodology should be based on fundamental principles of structural engineering.

One shortcoming of HyperLRFD++ is its slow performance. This inefficiency is partially due to Prolog's linear, top-down, and depth-first search method to deduce a conclusion. To improve the efficiency of the system, parallel processing seems promising for future implementation. In parallel logic programming, for example, two logic sentences representing results of "satisfied" and "violated" for a requirement are processed simultaneously, and conditions of each sentence are also processed in parallel instead of the top-bottom approach. Parallel execution of logic sentences would greatly enhance the performance of the system.

Another efficiency problem lies in the query processing in the HyperDocument model. The prototype HyperLRFD++ relies on a linear search method, i.e., the system searches from the first HyperTag to the desired cell one at a time. Query optimization should be addressed as a future work to enhance its performance. Query optimization is among the current research areas in HyperFile and hypertext technology [Clifton 90].

Although HyperTags in the HyperDocument model are hardware- and software-independent, contents of documents are hardware- and software-dependent. As noted in Section 2.2.2, SGML and HyTime are promising vehicles to make documents interchangeable among document processing systems. These new standards should be investigated in the future.

A limitation of the representation of design standards in the Object-Logic model is that if the organization of a design standard is unclear and ambiguous, the formulation of the Member Class Hierarchy depends on the developer of the system. In addition, design standards may contain a vague provision that could have multiple interpretations. This kind of provisions typically includes the modal auxiliary "may," such as "Second order effects may be considered in the determination of M_u for use in Formula H1-a and H1-b [AISC 86]." It may be interesting to apply many-valued logic [Ackerman 67] and modal logic [Moore 85] to this kind of information. Furthermore, there could exist a situation that the user may not be able to give an exact value to a data item but may be able to provide a fuzzy description such as "heavy," "tall," or "medium." A methodology to handle such "ambiguous" information is needed for design standards processing.

While the AISC specification is useful for designing components, there are other types of design standards that are needed for system design, such as the Uniform Building Code (UBC) [ICBO 91]. This type of standard is typically used to provide loads for a given system such as a building or a dam. Future research may include the integration of system design standards processing and system analysis systems to provide a structural system design model.

Last but not least, since the purpose of HyperLRFD++ is to demonstrate the viability of the model, its validation is limited to several textbook sample problems of structural steel design. Further evaluation should be performed for component design and conformance checking of structural members of a building structure.

References

[Ackerman 67] Ackerman, R. J., *An Introduction to Many-Valued Logics*, Dover Publications, 1967.

[AISC 86] *Manual of Steel Construction — Load & Resistance Factor Design*, First Edition, American Institute of Steel Construction, Inc., 1986.

[AISC 89] *Manual of Steel Construction — Allowable Stress Design*, Ninth Edition, American Institute of Steel Construction, Inc., 1989.

[Amble 87] Amble, T., *Logic Programming and Knowledge Engineering*, Addison-Wesley, 1987.

[Apple 90] *HyperCard Basics*, Apple Computer Inc., 1990.

[Bourdeau 91] Bourdeau, M., "The CD-REEF: The French Building Technical Rules on CD-ROM," *VTT Symposium 125, Computers and Building Regulations*, Espoo, Finland, Kahkonen, K. and Bjork, B. (eds.), pp.117-133, 1991.

[Ceri 90] Ceri, S, Gottlob, G., and Tanca, L., *Logic Programming and Databases*, Springer-Verlag, 1990.

[Chan 86] Chan, W. T., *Logic Programming for Managing Constraint-Based Engineering Design*, Ph.D. Thesis, Department of Civil Engineering, Stanford University, 1986.

[Clifton 90] Clifton, C. and Garcia-Molina, H., *Distributed Processing of Filtering Queries in HyperFile*, Technical Report No. CS-TR-295-90, Department of Computer Science, Princeton University, 1990.

[Conklin 87] Conklin, J., "Hypertext: An Introduction and Survey," *Computer*, Vol. 20, No. 9, pp.17-41, 1987.

[Cornick 91] Cornick, S. M., "HyperCode: The Building Code as a Hyperdocument," *Engineering with Computers*, Vol. 7, No. 1, pp.37-46, 1991.

[Cragun 87] Cragun, B. J. and Steudel, H. J., "A Decision-Table-Based Processor for Checking Completeness and Consistency in Rule-Based Expert Systems," *International Journal of Man-Machine Studies*, Vol. 26, No. 5, pp.633-648, 1987.

[Deville 90] Deville, Y., *Logic Programming*, Addison-Wesley, 1990.

[Dym 88] Dym, C. L., Henchey, R. P., Delis, E. A., and Gonick, S., "A Knowledge-based System for Automated Architectural Code Checking," *Computer-Aided Design*, Vol. 20, No. 3, pp.137-145, 1988.

[Elam 88] Elam, S. L. and Lopez, L. A., *Knowledge Based Approach to Checking Designs for Conformance with Standards*, Civil Engineering Systems Laboratory Research Series No. 9, University of Illinois at Urbana-Champaign, 1988.

[Fenves 66] Fenves, S. J., "Tabular Decision Logic for Structural Design," *Journal of the Structural Division,* Proceedings of ASCE, Vol. 92, No. ST6, 1966.

[Fenves 76] Fenves, S. J., Rankin, K., and Tejuja, H. K., *The Structure of Building Specifications*, National Bureau of Standards, Report No. NBSBSS-90, 1976.

[Fenves 77] Fenves, S. J. and Wright, R. N., *The Representation and Use of Design Specifications*, National Bureau of Standards, Report No. NBSTN 940, 1977.

[Fenves 85] Fenves, S. J. and Rasdorf, W. J., "Treatment of Engineering Design Constraints in a Relational Data Base," *Engineering with Computers*, Vol. 1, No. 1, pp.27-37, 1985.

[Fenves 87] Fenves, S. J., Wright, R. N., Stahl, F,I, and Reed,K. A., *Introduction to SASE: Standards Analysis, Synthesis, and Expression*, National Bureau of Standards, Report No. NBSIR 87-3513, 1987.

[Fischer 90] Fischer, G. and Nielsen, J., "Introduction to Hypertext and Hypermedia," Tutorial Text, *The Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[Garrett 86] Garrett, Jr., J. H. and Fenves, S. J., *A Knowledge-Based Structural Component Design*, Report No. R-86-157, Department of Civil Engineering, Carnegie-Mellon University, 1986.

[Garrett 89] Garrett, Jr., J. H., "An Object-Oriented Representation of Design Standards," *Proceedings of the Sixth Conference on Computing in Civil Engineering*, Atlanta, GA, pp.267–274, 1989.

[Genesereth 85] Genesereth, M. R. and Ginsberg, M. L., "Logic Programming," *Communication of the ACM,* Vol. 28, No. 9. pp.933-941, 1985.

[Genesereth 87] Genesereth, M. R. and Nilsson, N. S., *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann, 1987.

[Goldfarb 90] Goldfarb, C. F., *The SGML Handbook*, Oxford University Press, 1990.

[Harris 81] Harris, J. R. and Wright, R. N., *Organization of Building Standards: Systematic Techniques for Scope and Arrangement*, National Bureau of Standards, Report No. NBSBSS 136, 1981.

[Holtz 82] Holtz, N. M., *Symbolic Manipulation of Design Constraints*, Ph.D. Thesis, Department of Civil Engineering, Carnegie-Mellon University, 1982.

[ICBO 88] *Uniform Building Code*, 1988 Edition, International Conference of Building Officials, 1988.

[ICBO 91] *Uniform Building Code*, 1991 Edition, International Conference of Building Officials, 1991.

[Jain 89] Jain, D., Law, K. H., and Krawinkler, H., "On Processing Standards With Predicate Calculus," *Proceedings of the Sixth Conference on Computing in Civil Engineering*, Atlanta, GA, pp.259–266, 1989.

[Korson 90] Korson, T. and McGregor, J. D., "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, Vol. 33, No.9, pp.40-60, 1990.

[Kumar 89] Kumar, B., *Knowledge Processing for Structural Design*, Ph.D. Thesis, Department of Civil Engineering and Building Science, University of Edinburgh, 1989.

[Lakmazaheri 90] Lakmazaheri, S., *A Study on the Constraint Logic Approach for Structural Design Automation*, Ph.D. Thesis, Department of Civil Engineering, North Carolina State University, 1990.

[Malasri 91] Malasri, S., Olabe, J. C., and Olabe, M. A., "A Prototype of the Uniform Building Code Using Hypertext," *Proceedings of the Tenth Conference on Electronic Computation*, Indianapolis, IN, pp.36-43, 1991.

[Microsoft 91] *Microsoft Excel 3.0 Reference*, Microsoft Corporation, 1991.

[Mitusch 91] Mitusch, P. D., "Expert System for the Norwegian Building Regulations — A New Approach," *VTT Symposium 125, Computers and Building Regulations*, Espoo, Finland, Kahkonen, K. and Bjork, B. (eds.), pp. 36-42, 1991.

[Moore 85] Moore, R. C., "Semantical Considerations of Nonmonotonic logic," *Artificial Intelligence*, Vol. 25, No. 1, pp.75-94, 1985.

[Newcomb 91] Newcomb, S. R., Kipp, N. A., and Newcomb, V. T., "The 'HyTime' Hypermedia/Time-based Document Structuring Language," *Communications of the ACM*, Vol. 34, No. 11, pp.67-83, 1991.

[Nguyen 87] Nguyen, T. A., Perkins, W. A., Laffey, T. J., and Pecora, D., "Knowledge Base Verification," *AI Magazine*, Vol. 8, No. 2, pp.69-75, 1987.

[Oracle 89] *ORACLE for Macintosh Reference*, Oracle Corporation, 1989.

[Owl 90] *GUIDE*, OWL International, 1990.

[Page 89] Page, T. W., *An Object-Oriented Logic Programming Environment for Modeling*, Ph.D. Thesis, University of California at Los Angeles, 1989.

[Parsaye 89] Parsaye, K., Chignell, M., Khoshafian, S., and Wong, H., *Intelligent Databases*, John Wiley & Sons, Inc., 1989.

[Quintus 90] *Quintus Prolog++ Reference Manual*, Quintus Computer Systems, Inc., 1990.

[Quintus 91] *MacDBI for Oracle Reference Manual*, Quintus Computer Systems, Inc., 1991.

[Rasdorf 88] Rasdorf, W. J. and Wang, T. E., "Generic Design Standards Processing in an Expert System Environment," *Journal of Computing in Civil Engineering*, Vol. 2, No. 1, pp.68–87, 1988.

[Rasdorf 90-a] Rasdorf, W. J. and Lakmazaheri, S., "Logic-Based Approach for Modeling Organization of Design Standards," *Journal of Computing in Civil Engineering,* Vol. 4, No.2, pp.102–123, 1990.

[Rasdorf 90-b] Rasdorf, W. J. and Lakmazaheri, S., "A Logic Based Approach for Processing Design Standards," *International Journal of Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, Vol. 4, No. 3, pp.179-192, 1990.

[Rokach 91] Rokach, A. J., *Theory and Problems of Structural Steel Design (Load and Resistance Factor Method)*, Schaum's Outline Series, McGraw-Hill, 1991.

[Rosenman 85] Rosenman, M. A. and Gero, J. S., "Design Codes as Expert Systems," *Computer-Aided Design*, Vol. 17, No. 9, pp.399–409, 1985.

[Rosenman 86] Rosenman, M. A., Gero, J. S., and Oxman, R., "An Expert System for Design Codes and Design Rules," *Applications of Artificial Intelligence in Engineering Problems*, Vol. II, Sriram, D and Adey, R (eds.), Springer-Verlag, pp.745-758, 1986.

[Salmon 90] Salmon, C. G. and Johnson, J. E., *Steel Structures: Design and Behaviors*, Third Edition, Harper & Row, 1990.

[Stone 87] Stone, D. and Wilcox, D. A., "Intelligent Systems for the Formulation of Building Regulations," *Proceedings of the Fourth International Symposium on Robotics and Artificial Intelligence in Building Construction*, Haifa, Israel, pp.740-761, 1987.

[Suwa 82] Suwa, M., Scott, A. C., and Shortliffe, E. H., "An Approach to Verifying Completeness and Consistency in Rule-Based Expert System," *AI Magazine*, Vol. 3, No. 4, pp.16-21, 1982.

[Swartout 83] Swartout, W. R., "XPLAIN: a System for Creating and Explaining Expert Consulting Programs," *Artificial Intelligence*, Vol. 21, No. 3, pp.285-325, 1983

[Van Herwijnen 90] Van Herwijnen, E., *Practical SGML*, Kluwer Academic Publishers, 1990.

[Yura 71] Yura, J. A., "The Effective Length of Columns in Unbraced Frames," *Engineering Journal*, Vol. 8, No. 2, pp.37-42, 1971.