CONSTRAINT PROCESSING ALTERNATIVES IN AN ENGINEERING DESIGN DATABASE

by

Michael J. Soha^far

D^osmbar, 1922

DRC-01-13-32

CONSTRAINT PROCESSING ALTERNATIVES IN AN ENGINEERING DESIGN DATABASE

By

Michael Joseph Schaefer

B.S.C.E - Carnegie-Mellon University, 1981

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Department of Civil Engineering

Carnegie-Mellon University

Pittsburgh, Pennsylvania

October 28, 1982

ABSTRACT

Methodologies have been developed to provide assistance in the maintenance of integrity and consistency within a relational database. These methodologies usually entail the complete removal of all dependencies between the data items. A proposed engineering design database model provides a *constraint processing* mechanism that is capable of checking the satisfaction of specific dependencies or constraints. The objective of this thesis is to expand this model so that it can be implemented in a large engineering design relational database system.

An introduction to the basic issues of database management and to the concepts of the relational database model are presented. Examples of basic data retrieval operations and an actual relational data retrieval language are provided. Guidelines for database design are also developed.

Basic constraint processing issues including a description of the different constraint sources and types are presented. A proposed constraint processing mechanism is described and extended to allow the enforcement of a wide range of constraint types and provide the flexibility needed for engineering design.

An actual design example is used to provide a detailed comparison between the extended mechanism and the traditional approach to constraint enforcement.

KEYWORDS: database; relational database; design database; data integrity; data consistency; **information retrieval;** queries; normalization; **constraints;** constraint processing; constraint enforcement; **computer programs; structural engineering; engineering design;**

ACKNOWLEDGMENTS

The author wishes to express his gratitude to Dr. Daniel FL Rehak for the guidance and assistance he provided in the development of this thesis.

The author is especially indebted to Dr. Steven J. Fenves for the valuable advice and direction as well as the patience he so frequently showed throughout the author's undergraduate and graduate career.

Thanks are also extended to the staff and students of the Civil Engineering Department for helping to make this graduate program a successful and enjoyable experience.

The author also extends his love and appreciation to his wife, Renay, and his family for their support and for providing the necessary distractions that made this thesis possible.

This research was funded by the National Science Foundation under grant PER-8013746, entitled "Software Aids for Design Specification Development and Use."

Information in Chapter 2 has been submitted for publication to the Technical Council of Computing Practices of the American Society of Civil Engineers (ASCE).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE
INTRODUCTION

1.1. BACKGROUND

During the design and construction of any engineering project, large amounts of information pertaining to all aspects of the project must be stored, accessed and operated upon. One way to store this information is in a database. A database offers two main advantages. First, it provides *data independence.* The way in which data is stored or accessed in the database is independent from its use. The user's view of the data is defined by the *conceptual schema* which provides the definition of information content only. No references to storage/access details are included in the schema [9]. The user need only specify which data items are needed, not how to find them or where to look for them. Therefore, even if one user needs the data in one form and another user needs the same data in a different form, the data is stored only once. It is then the database system's responsibility to supply the data in the appropriate form. The user specifies what data is needed through a retrieval language. An example of such language is shown in Section 2.3.2. Most database systems offer access to the retrieval operations from two different modes. First, the *standalone* mode provides the user with interactive access to the data. The second mode is the *application program* mode. It supplies the retrieval operations as a data sublanguage which can be called by a host programming language in an application program.

A second advantage that a database provides is *centralized control* of the data [9], allowing all users to access the same data. Instead of each user having his own private files, which would probably store data similar to other user's files, all fifes can be integrated into one database, thus eliminating redundancy. Since the data are stored in only one place, centralized control also insures that the data are consistent. Once a piece of data is changed, all

users access the most recent value. Centralized control also eases the process of insuring *data integrity,* that is, that all updates of the data are valid. Whenever an update is attempted, a predefined checking procedure based on a specific *constraint* can be invoked to determine the validity of the update.

1.2. OBJECTIVES

Since "a major aspect of engineering design involves the evaluation and satisfaction of constraints" [15], it is important that the database be able to process these constraints to insure integrity and consistency of the data contained within. In order to insure complete integrity and consistency of the data, the constraint processing mechanism must be flexible so that the designer can specify which constraints need to be satisfied and so that a wide range of constraint types can be enforced.

Therefore, the primary objectives of this thesis are to:

- investigate the use of constraints and constraint processing mechanisms to insure integrity and consistency in an engineering design database;
- extend a constraint processing mechanism proposed by Rasdorf [19] so that it can be implemented for a wide range of constraint types; and
- provide a design example comparing the presented mechanism to the traditional approach to constraint processing.

1.3. SURVEY OF PREVIOUS WORK

Until recently, most research in the area of constraint processing has centered around the normalization process which removes dependencies from and therefore insures integrity within the database. Codd [7] introduced this process by describing the basic *normal forms* or database configurations that remove a number of common dependencies. Other normal forms have been presented by Fagin [10], [11], Aho, Beeri and Ullman [1] and others. Additional work in the area of constraint enforcement through normalization includes a method for testing the database schema [18] and a proposed algorithm for viewing the dependencies and deciding upon a proper configuration within the database [12] so that integrity is maintained.

As stated above, the normalization process eliminates dependencies so integrity is maintained. It appears to provide a successful integrity enforcement mechanism for the business-oriented applications for which it is developed. However, engineering design applications require greater flexibility and control over the constraints. Instead of eliminating a dependency within a database, it may be more useful to store the dependent data and insure integrity by introducing a constraint checking mechanism into the database system. Stonebraker [21] described a constraint processing mechanism which can enforce four different types of constraints. Bernstein, Blaustein and Clarke [3] introduced a mechanism that stores specific information about the data (such as minimum and maximum values) so that updates to the database can be checked against this *aggregate data* to find possible integrity violations.

Although these methods allow constraint checking, they do not provide the necessary flexibility for engineering design. Rasdorf [19] proposed a constraint processing mechanism that provides the user control over the particular constraints to enforce and stores the status (satisfied or violated) of the constraint at the time the constraint is checked. This mechanism, however, can not handle the large selection of constraint types as discussed in [21].

1.4. ORGANIZATION

Chapter 2 provides an introduction to the basic concepts of database management. The relational database model is presented, along with a discussion of data retrieval and normalization issues.

Basic constraint processing issues including a description of constraint sources and types is presented in Chapter 3. A comparison is provided between traditional constraint enforcement and the constraint processing mechanism proposed by Rasdorf [19]. Extensions to this mechanism are described which enable it to handle a larger number of constraint types.

Chapter 4 outlines a design example which is used in Chapter 5 to continue the comparison between traditional constraint enforcement and the extended version of Rasdorf's work. Data items, constraints and design processes are presented, along with the basic database schema.

A detailed comparison between a normalized design database and a complete constraint checking database system which includes relations that are augmented with constraint status attributes is presented in Chapter 5.

Finally, Chapter 6 summarizes the results of the investigation. Specific conclusions are listed and potential areas of future work are presented.

CHAPTER TWO
RELATIONAL DATABASE CONCEPTS

This chapter provides an introduction to relational database management systems (RDBMS). Basic concepts are discussed, including data retrieval and normalization.

2.1. WHY A RELATIONAL DATABASE?

A relational database stores information in a set of relations, which are basically two-dimensional tables. All information about each aspect of the project can be stored in a set of one or more relations. An example of a relational database, used throughout this chapter, is shown in Figure 2-1. This database is made up of three relations; BEAMS, DESIGNATIONS and GRADES. The BEAMS relation is similar to a beam schedule for a construction project; it lists the LENGTH, DESIGNATION, GRADE of steel and quantity (QTY) of each specific shape and grade in the project. The DESIGNATIONS relation is similar to the structural shape tables in the AISC manual [2] and contains the DESIGNATION, SECTION MODULUS, AREA, moment of inertia (I) and depth (D) for each shape. Finally, the GRADES relation lists the GRADE and yield stress (FY) for different grades of steel.

Since, as stated by Codd [8], "relational database technology offers dramatic improvements in productivity", relational databases are of great interest to the developers of engineering design databases. Therefore, using examples similar to the database in Figure 2-1, this chapter will provide an introduction to the relational database approach by discussing some of the basic concepts involved, including guidelines for database design through normalization.

BEAMS

LENGTH	DESIGNATION	GRADE	QTY
20	W36x300	A36	15
20	W36x300	A514	9
20	W33x241	A514	5
20	W33x241	A36	5
40	W30x211	A588	8
40	W30x211	A242	4
20	W27x114	A36	2
35	W16x57	A36	3
20	W27x114	A514	7

DESIGNATIONS

DESIGNATION	SECTION MODULUS	AREA	I	D
W36x300	1110	88.3	20300	36.74
W33x241	829	70.9	14200	34.18
W30x211	663	62.0	10300	30.94
W27x114	299	33.5	4090	27.29
W16x57	92.2	16.8	758	16.43

GRADES

GRADE	Fy
A36	36
A514	100
A588	50
A242	50

Figure 2-1: Example of a Relational Database.

2.2. NOMENCLATURE

This section will describe a few of the basic terms in the relational database model.

The columns of the relations in Figure 2-1, such as LENGTH, DESIGNATION, SECTION MODULUS and GRADE, are called *attributes.* The actual values for each attribute are taken from the *domain* of the attribute. A domain is the set of all allowable values for a specific attribute. For example, the quantity (QTY) attribute in the BEAMS relation (Figure 2-1) has for its domain all integers greater than zero. In any given relation, the ordering of the columns is arbitrary.

The individual rows of a relation are called *tuples.* No tuple is duplicated within any relation. A tuple thus represents some unique object or entity with its properties (attribute values). As is the case with domains, the ordering of the tuples within a relation is also arbitrary.

In each relation, a group of one or more of the attributes uniquely identifies each tuple. This group of attributes is called the *key* for the relation. The key for the BEAMS relation in Figure 2-1 is the attribute combination of LENGTH, DESIGNATION and GRADE: with these attributes specified, a specific tuple is uniquely determined. For the DESIGNATIONS relation, the key is DESIGNATION. The key for the GRADES relation is GRADE.

The overall configuration of the database is referred to as the *schema.* The schema is the layout of the relations themselves, without the data. The description of an individual relation is called the *relation schema.* Once all relations are described, such as the attribute names and types in each relation, the database schema is completely defined.

A *constraint* is a tool used to control the integrity of a relation. As stated above, the quantity (QTY) attribute in the BEAMS relation must be greater than zero. Therefore, the following constraint is specified:

QTY > 0

Also, if the value of one attribute is dependent upon another, a constraint can be specified to insure this dependency. For example, if a relation contains the

height, width and area of a solid rectangular beam, the user may want to specify the following constraint:

AREA = WIDTH X HEIGHT

If a user tries to change one of these attribute values for a tuple without changing any other value, the database system would disallow the change and warn the user of the constraint violation.

2.3. **DATA RETRIEVAL**

Once a database is created, any authorized user must be able to access the data. Section 2.3.1 will discuss the three basic relational operators that support this process: *SELECT, PROJECT* and *JOIN.* Section 2.3.2 will demonstrate the implementation of these basic operators in a current relational database management system query language.

The examples in this section show the DBMS commands in **BOLD FACE,** the relation names in UPPER CASE and the attribute names in lower case letters.

2.3.1. BASIC OPERATORS

This section presents the three basic relational operators. These three operators can be used individually or in any combination to retrieve data from the database. The PROJECT and JOIN operators will be used in Section 2.4 to describe the normalization process.

2.3.1.1. SELECT

The **SELECT** operator is used to produce a new relation from the rows of an existing relation for which a specified selection criterion is satisfied. The criterion is specified by attaching a boolean predicate to the **SELECT** command through the use of a **WHERE** clause. All tuples for which the **WHERE** clause evaluates to true are retrieved. For example, the following command forms a new relation from the BEAMS relation of Figure 2-1, containing all beams whose length equals 40 feet.

SELECT BEAMS **WHERE** length = 40

The result is:

LENGTH	DESIGNATION	GRADE	QTY
40	W30X211	A588	8
40	W30X211	A242	4

2.3.1.2. PROJECT

The **PROJECT** operator is used to produce a new relation containing one or more of the columns of an existing relation. For example, the following command forms a new relation from the BEAMS relation in Figure 2-1, by eliminating the GRADE and quantity (QTY) columns.

PROJECT BEAMS **OVER** length **AND** designation

The result is:

LENGTH	DESIGNATION
20	W36X300
20	W33X241
40	W30X211
20	W27x114
35	W16X57

Notice that the duplicate tuples created by the projection are removed.

2.3.1.3. JOIN

The **JOIN** operator is used to produce a new relation by combining two or more relations over a comparison between attributes with common domains. Therefore, the relations to be joined must contain a common domain. For example, the following command forms a new relation containing the attributes LENGTH, DESIGNATION, GRADE and quantity (QTY) from the BEAMS relation in Figure 2-1 and the attributes GRADE and FY from the GRADES relation in Figure 2-1.

JOIN BEAMS **AND** GRADES OVER grade

The result is:

LENGTH	DESIGNATION	GRADE	QTY	GRADE	Fy
20	W36X300	A36	15	A36	36
40	W36X300	A514	9	A514	100
40	W33x241	A514	5	A514	100
20	W33x241	A36	5	A36	36
40	W30X211	A588	8	A588	50
40	W30X211	A242	4	A242	50
20	W27X114	A36	2	A36	36
35	W16X57	A36	3	A36	36
20	W27X114	A514	7	A514	100

This is an example of an *equijoin;* the two relations are joined where the grade values are equal. Other types of joins are possible, such as, *greater than* and *less than* joins. These joins combine the relations where a specific attribute in one relation is greater than (or less than) an attribute in another relation.

Also, notice that the equijoin produces a relation with two identical columns. This redundancy must be eliminated by projecting out one of the duplicate attributes. However, since the removal of the duplicate attribute is a very common operation after a join, a *natural join* can be specified, which is the same as an equijoin with the duplicate attribute eliminated.

2.3.2. **IMPLEMENTATION OF RELATIONAL OPERATORS**

Many languages have been developed to ease the data retrieval operations for the inexperienced user. One such language is called SEQUEL (Structured English QUEry Language) [5]. After many extensions and improvements, SEQUEL 2 was developed [6] and implemented in the Database Management System (DBMS) entitled SYSTEM R [4]. This language is used in the following examples to query, manipulate, define and control the database in Figure 2-1.

2.3.2.1. QUERYING **THE DATABASE**

The query facility of SEQUEL 2 allows the user to ask the database questions about the data. The **SELECT** command is used to specify which attributes are to be returned. The **FROM** command is used to specify the relation to use for the **SELECT** command. The **WHERE** command is used to compare attributes to values or to other attributes. The comparisons may be connected by the operators **AND** and **OR** to form more complex comparisons. Also, the result of a query may be used in the **WHERE** command of another query. The following examples are typical queries.

1. List the DESIGNATION, depth (D) and SECTION MODULUS of all beams with a depth less than 34 inches, sorted in order of increasing section modulus.

```
SELECT designation, D, section modulus
FROM DESIGNATIONS
WHERE D < 34
ORDER BY section modulus
```

The result is:

DESIGNATION	D	SECTION MODULUS
W16X57	16.43	92.2
W27X114	27.29	299
W30X211	30.94	663

The **ORDER BY** command orders the tuples of a relation by the attribute specified. However, the ordering is done specifically for the user and does not effect the actual storage of the relation, since the order of tuples is completely arbitrary.

2. List the average area of the 20 foot beams.

```
SELECT AVERAGE(area)
FROM DESIGNATIONS
WHERE designation IN
        SELECT designation
        FROM BEAMS
```

WHERE length = 20

The result is:

AREA
64.2

This command is the same as the following list of operations using the basic operators, defined in Section 2.3.1.

- **SELECT** BEAMS **WHERE** (length.**EQ**.20)
- **JOIN** RESULT **AND** DESIGNATIONS **OVER** designation
- **PROJECT** RESULT **OVER** area
- Compute average area using the result of the above command.

3. Find all beams in the project made of A514 steel.

```
SELECT UNIQUE designation
FROM BEAMS
WHERE grade = 'a514'
```

The result is:

DESIGNATION
W36x300
W33x241
W27x114

Notice that the word 'unique' follows the select command. This is because SEQUEL 2 allows duplicate tuples and only removes them upon request.

4. Find the yield stress for all W36x300 beams in the project.

```
SELECT UNIQUE Fy
FROM GRADES
WHERE grade IN
```

```
SELECT grade
FROM BEAMS
WHERE designation = 'W36x300'
```

The result is:

FY
36
100

5. List the DESIGNATION, SECTION MODULUS and the values of 2«I/D for all beams with a 2*I/D value less than 830. Notice that the derived value of 2*I/D is calculated by the database system.

```
SELECT designation, section modulus, 2*I/D
FROM beams
WHERE 2«I/D < 830
```

The result is:

DESIGNATION	SECTION MODULUS	2*I/D
W30X211	663	665.80
W27x114	299	299.74
W16X57	92.2	92.27

6. How many different beam designations are in the project?

```
SELECT COUNT (UNIQUE designation)
FROM BEAMS
```

The result is:

5

2.3.2.2. MANIPULATION OF THE DATABASE

The manipulation aspect of SEQUEL 2 allows the user to change, add and/or delete values in the database, as shown in the following examples.

1. Insert a new beam into the BEAMS relation with a length of 60 feet, a designation of W27x114 and a grade of A36, leaving the quantity null until it can be determined.

```
INSERT INTO BEAMS (length, designation, grade);
                  <60, W27x114, A36>
```

This command causes the system to create a new tuple in the BEAMS relation with the given data in the appropriate attribute columns. Notice that the attributes to be stored are listed after the relation name. If all attributes are to receive values, the list is optional.

2. Create a new relation called SMALL DEPTHS, with the same attribute names as DESIGNATIONS, that includes beams from the DESIGNATIONS relation that have depth values less than 30.

```
ASSIGN TO SMALL DEPTHS (designation, section modulus, area,
                        I, D);
        SELECT <designation, section modulus, area, I, D>
        FROM DESIGNATIONS
        WHERE D < 30
```

The following relation is created by this command:

SMALL DEPTHS

DESIGNATION	SECTION MODULUS	AREA	I	D
W27x114	299	33.5	4090	27.29
W16x57	92.2	16.8	758	16.43

3. Add to the SMALL DEPTHS relation all beams in the DESIGNATION relation with depths less than 34 (for this example it is assumed that the beams with depths less than 30 are already contained in the SMALL DEPTHS relation).

```
INSERT INTO SMALL DEPTHS
            SELECT (designation, section modulus, area, I, D)
            FROM DESIGNATIONS
```

WHERE D < 34 AND D > 30

The result is:

SMALL DEPTHS

DESIGNATION	SECTION MODULUS	AREA	i	D
W27x114	299	33..5	4090	27
W16X57	92.2-	16.	758	16.43
W30X211	663	62.0	10300	30.94

4. Delete all beams in the SMALL DEPTHS relation with depths greater than 30.

```
DELETE SMALL DEPTHS
        WHERE D > 30
```

This command would return the SMALL DEPTHS relation to its original form.

5. Update the BEAMS relation by adding 10 feet to the beams with a length equal to 35 feet.

```
UPDATE BEAMS
SET length * length + 10
WHERE length = 35
```

This command changes the eighth tuple in the BEAMS relation to:

45	W16X57	A36	3

2.3.2.3. DATA DEFINITION

The data definition aspect of SEQUEL 2 allows the user to create, delete and/or change the structure of relations in the database.

1. The following command would be used to define (create) the DESIGNATIONS relation:

```
CREATE TABLE DESIGNATIONS <designation(CHAR<7), NONULL),
                section modulus(DECIMALd)), area(DECIMAL(1»,
                I(DECIMAL(1», D(DECIMAL(2)))
```

The CHAR(n) specification means that the value for the appropriate

attribute is always a character string of at most n characters. **NONULL** means that the attribute must always be specified for each tuple. **DECIMAL(n)** means that the value for the attribute is a real number with at most n decimal places.

2. Add an attribute (column) to the designation relation to store the web thickness (TW).

   EXPAND COLUMN TW(DECIMAL(3»

3. Delete the GRADE relation.

   DROP TABLE GRADE

2.3.2.4 CONTROL OF THE DATABASE

This aspect of SEQUEL 2 allows the users to control the access of their data to other users and to exercise control over the integrity of data values. Access is controlled by the **GRANT** and **REVOKE** commands. Data integrity is controlled by specifying assertions on the data, using the **ASSERT** command. Each assertion is given a name by the user who specifies it and is referenced by this name whenever it is violated.

1. Allow Smith complete access (read, insert and change) to the BEAMS relation, including the option to give access to someone else.

   GRANT READ, INSERT, UPDATEdength, designation, grade, qty)
   ON BEAMS **TO** Smith **WITH GRANT OPTION**

2. Disallow Smith's access to the BEAMS relation.

   REVOKE ALL RIGHTS ON BEAMS **FROM** Smith

   The **ALL RIGHTS** command can always be substituted by a list of rights as seen in the above example.

3. Require that all quantities in the BEAMS relation are greater than zero.

   ASSERT A1 **ON** BEAMS: qty > 0.0

4. Require all beam lengths in the BEAMS relation to be greater than 5 but less than 100.

   ASSERT A2 **ON** BEAMS: length **BETWEEN** 5 **AND** 100

5. Require each beam in the BEAMS relation to have a designation equal to one of the designations in the DESIGNATIONS relation.

ASSERT A3:
(SELECT designation **FROM** BEAMS)
IS IN
(SELECT designation **FROM** DESIGNATIONS)

2.4. NORMALIZATION

Normalization is a method for insuring that the organization of the database has certain desirable properties. It is a tool for determining the best possible arrangement of the data in the database. Six different normal forms will be discussed; *first, second, third fourth, fifth,* and *domain-key* normal form. The description of first, second, third, fourth and fifth normal forms is based on the discussion of normalization by Date [9].

2.4.1. FIRST NORMAL FORM

All relations in a relational database must be in first normal form (INF). There are two reasons for this: first, storage is easier, and second, the data manipulation operators are not as complicated as they would be for unnormalized relations. A relation not in 1NF is called an unnormalized relation. Figure 2-2 is an example of an unnormalized relation.

BEAM INFORMATION			Fy	SECTION MODULUS	AREA	ID	QTY
LENGTH	DESIGNATION	GRADE					
20	W36X300	A36 A514	36 100	1110	88.3	20300 36.74	15 9
20	W33x241	A514 A36	100 36	829	70.9	14200 34.18	5
40	W30X211	A588 A242	50 50	663	62.0	10300 30.94	Q 4
35	W16X57	A36	36	92.2	16.8	758 16.43	3
20	W27X114	A36 A514	36 100	299	33.5	4090 27.29	2 7

Figure 2-2: Relation BEAM: an unnormalized relation.

The key for this relation, beam information, is made up of the sub-attributes

LENGTH, DESIGNATION and GRADE. This relation is considered unnormalized, because not all rows and columns contain single values. Figure 2-3 shows an equivalent normalized relation. The key for this relation is a composite key consisting of LENGTH, DESIGNATION and GRADE. Notice that all values are simple, that is, they are all single valued.

LENGTH	DESIGNATION	GRADE	Fy	SECTION MODULUS	AREA	I	D	QTY
20	W36X300	A36	36	1110	88.3	20300	136.74	15
20	W36X300	A514	100	1110	88.3	20300	136.74	9
20	W33x241	A514	100	829	70.9	14200	134.18	5
20	W33x241	A36	36	829	70.9	14200	134.18	5
40	W30X211	A588	50	663	62.0	10300	130.94	8
40	W30X211	A242	50	663	62.0	10300	130.94	4
35	W16X57	A36	36	92.2	16.8	758	16.43	3
20	W27X114	A36	36	299	33.5	4090	27,29	2
20	W27X114	A514	100	299	33.5	4090	27.29	7

Figure 2-3: Relation Beami: first normal form.

This leads to the definition of a normalized relation:

> A relation whose attributes or tuples cannot be broken down into two or more attributes or tuples is considered normalized.

2.4.2. SECOND NORMAL FORM

After examining the relation in figure 2-3, one may observe that certain problems arise because of the *partia/* dependency of the non-key attributes on the attributes in the composite key. As shown in figure 2-4, Fy is uniquely determined by GRADE. Also, SECTION MODULUS, AREA, I and D are uniquely determined by DESIGNATION. These partial dependencies exist even though the entire key of LENGTH, GRADE and DESIGNATION is necessary to identify a specific tuple. Notice also that section modulus is determined by I and D. This is called a transitive dependency and will be described in Section 2.4.3. The partial dependencies cause the following problems:

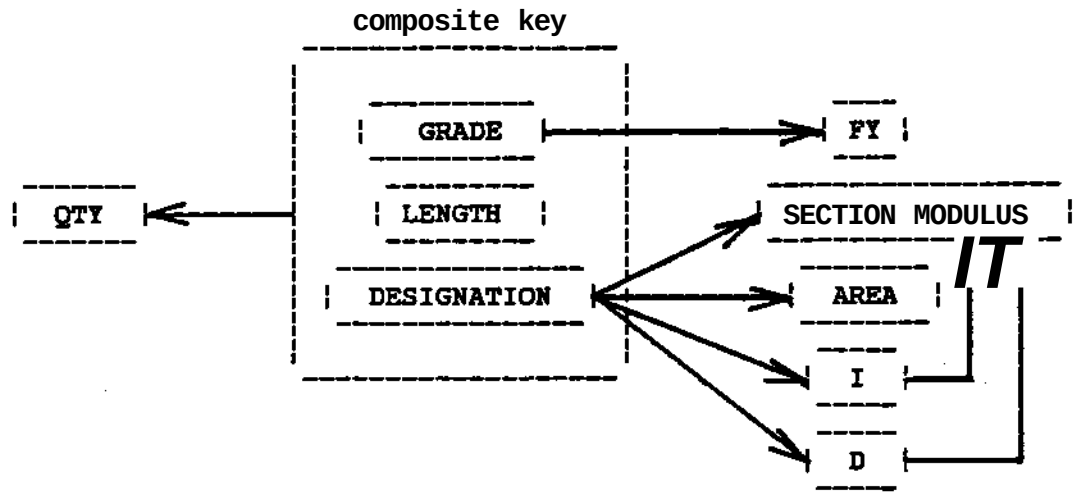

Figure 2-4: Dependencies in BEAM1 relation.

- **There is no way to store the yield stress (Fy) of a grade A529 steel since none of the beams are made of grade A529 steel.**
- **If the W16x57 beam is deleted, the relation no longer contains the I, D, area or section modulus for this shape.**
- **If it was discovered, after the data was entered, that the area of the W36x300 beam is not 88.3, then all tuples would have to be searched to find all occurrences of W36x300 to correct the error.**

These problems are eliminated by creating three new relations from the BEAM1 relation. As shown in Section 2.3.1.2, this can be done by the PROJECT operator. Figure 2-5 shows these new relations. With the database in this form, all of the previous problems are solved.

- **Any beam designation or grade can be stored, even if those designations or grades are not currently in the BEAM2 relation.**
- **A particular beam occurrence can be deleted, without losing its related dimensions.**
- **Information about any designation or grade can be changed without having to search for multiple occurrences.**

By eliminating the partial dependencies on the key, the relations in figure 2-5 are now considered to be in at least second normal form[1] (2NF). The definition of second normal form is then as follows:

[1] Actually, the BEAM2 **and** GRADE relations **are** also in third and fourth normal forms.

BEAM2

LENGTH	DESIGNATION	GRADE	QTY
20	W36x300	A36	15
20	W36x300	A514	9
20	W33x241	A514	5
20	W33x241	A36	5
40	W30x211	A588	8
40	W30x211	A242	4
20	W27x114	A36	2
35	W16x57	A36	3
20	W27x114	A514	7

DESIGNATION

DESIGNATION	SECTION MODULUS	AREA	I	D
W36x300	1110	88.3	20300	36.74
W33x241	829	70.9	14200	34.18
W30x211	663	62.0	10300	30.94
W27x114	299	33.5	4090	27.29
W16x57	92.2	16.8	758	16.43

GRADE

GRADE	Fy
A36	36
A514	100
A588	50
A242	50

Figure 2-5: BEAM2, DESIGNATION and GRADE relations: second normal form.

A 1NF relation is in second normal form if all attributes depend on the entire key.

2.4.3. THIRD NORMAL FORM

Problems can also arise in a relation in second normal form, due to the possibility of *transitive* dependencies of two or more non-key attributes. An example of a transitive dependency is shown in Figure 2-6 for the DESIGNATION relation in Figure 2-5.

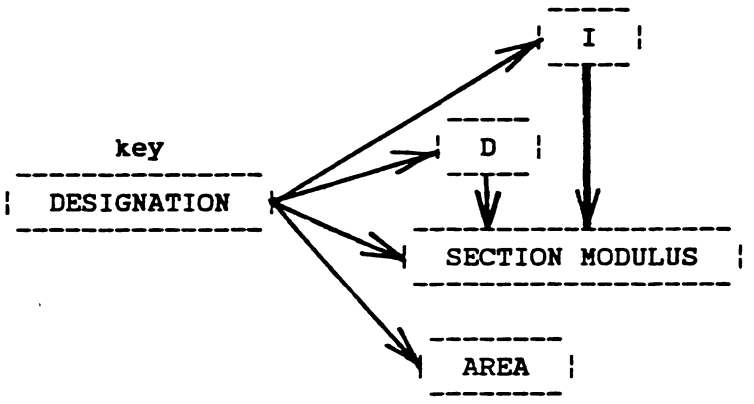

Figure 2-6: Transitive dependencies between I, D and section modulus.

Although all attributes depend on the key, DESIGNATION, another dependency exists between SECTION MODULUS, I and D, since, theoretically, section modulus equals I/(D/2). This kind of transitive dependency can cause problems similar to the ones in a relation in 1NF. However, in this case none of these problems are present. This is because a SECTION MODULUS relation containing SECTION MODULUS (as the key for the relation), I and D could not stand alone; two beam designations could have the same section modulus even if the I's and D's were different, thus forcing the key of SECTION MODULUS to be non-unique. Ignoring the lack of an actual problem in this particular relation, normalization would force it to be split, so that the transitive dependency is removed. Figure 2-7 shows two new relations that remove this dependency. These new relations are in at least third normal form.[2]

---

[2] Actually, these relations are also in fourth normal form, as explained in Section 2.4.4.

DESIGNSA

DESIGNATION	SECTION MODULUS	AREA
W36X300	1110	88.3
W33x241	829	70.9
W30X211	663	62.0
W27x114	299	33.5
W16X57	92.2	16.8

DESIGNID

DESIGNATION	j	D
W36X300	20300	36.74
W33X241	14200	34.18
W30X211	10300	30.94
W27X114	4090	27.29
W16X57	758	16.43

Figure 2-7: DESIGNSA and DESIGNID relations: third normal form.

This leads to the definition of third normal form.

A 2NF relation is in third normal form if all attributes depend on the key and no other attribute.

2.4.4. FOURTH NORMAL FORM

A close look at the BEAM2 relation in figure 2-5 uncovers another problem. To provide a clearer example of this problem, a new relation similar to BEAM2 is created by removing the quantity attribute and changing a few tuples. The new relation (BEAM3) is *all key.* since all attributes must be specified to uniquely identify any specific tuple. Figure 2-8 shows the new relation. Assume that this relation is a list of all the possible types of beams that can be provided. This means that only 20 and 40 feet lengths are allowed: that a 20 foot beam can only have a grade of A36 or A514, and that the 40 foot beams can only be made of A588 or A242 steel. This dependency between LENGTH and GRADE is called a *multi-valued* dependency (MVD). A multi-

BEAM3

LENGTH	DESIGNATION	GRADE
20	W36X300	A36
20	W36X300	A514
20	W33x241	A514
20	W33X241	A36
40	W30X211	A588
40	W30X211	A242
20	W27x114	A36
20	W27X114	A514
20	W30X211	A36
20	W30X211	A514

Figure 2-8: BEAM3 relation.

valued dependency is similar to the transitive dependency: in a relation with a transitive dependency, one or more attributes uniquely determine the value of another attribute, while in a relation with a multi-valued dependency, one or more attributes determine a well-defined set of values for another attribute.

This multi-valued dependency causes the following two problems:

- There is a large amount of redundancy in this relation. This redundancy causes the same updating problem as before; if a designation is incorrect, all occurrences have to be searched and changed.

- If a new grade of steel can be provided for 20 foot beams, three new tuples must be entered, one for each distinct designation. This adds to the redundancy.

The MVD is eliminated by creating two new relations. The new relations are shown in Figure 2-9. These relations are now considered to be in fourth normal form (4NF), which is defined in the following way:

> A 3NF relation is in fourth normal form if, when a multi-valued dependency exists, one attribute upon another, then all other attributes depend on this same attribute.

DESIGNATION!,

LENGTH	DESIGNATION
20	W36X300
20	W33X241
20	W27X114
40	W30X211
20	W30X211

GRADEL

LENGTH	GRADE
20	A36
20	A514
40	A588
40	A242

Figure 2-9: DESIGNATiONL and GRADEL relations: fourth normal form.

2.4.5. FIFTH NORMAL FORM

After examining the discussion on fourth normal form in the previous section, it could be decided to ignore the redundancy problem in the BEAM3 relation shown in Figure 2-8. If this is done, the BEAM3 relation will remain intact and the user has to be aware of the redundancy. This is a possible alternative, since any relation that is at least in first normal form is a valid relation. However, there is a different type of problem associated with a relation like the BEAM3 relation that could cause more than just a redundancy problem. This new problem arises from the *join dependency* of the attributes. To show this join dependency, assume that, at some time, the BEAM3 relation is broken into its three projections: DESIGNATIONUcontaining the length and designation), GRADEUcontaining the length and grade) and DESIGRADE(containing designation and grade). Then, at some other time, only two of these relations, such as DESIGNATIONL and DESIGRADE, are joined to form the original relation. Figure 2-10 shows that this process causes four new tuples to appear in the new BEAM3 relation (called NBEAM3) that did not appear in the original BEAM3 relation. This is called a join dependency. It is only when the third projection is also joined that the NBEAM3 relation is the same as the original BEAM3 relation. Therefore, in order to eliminate this join dependency, the BEAM3 relation should be broken into three new relations, each with a different key corresponding to the number of possible or "candidate" keys in BEAM3, and left in this form.

Fifth normal form can be defined in the following way:

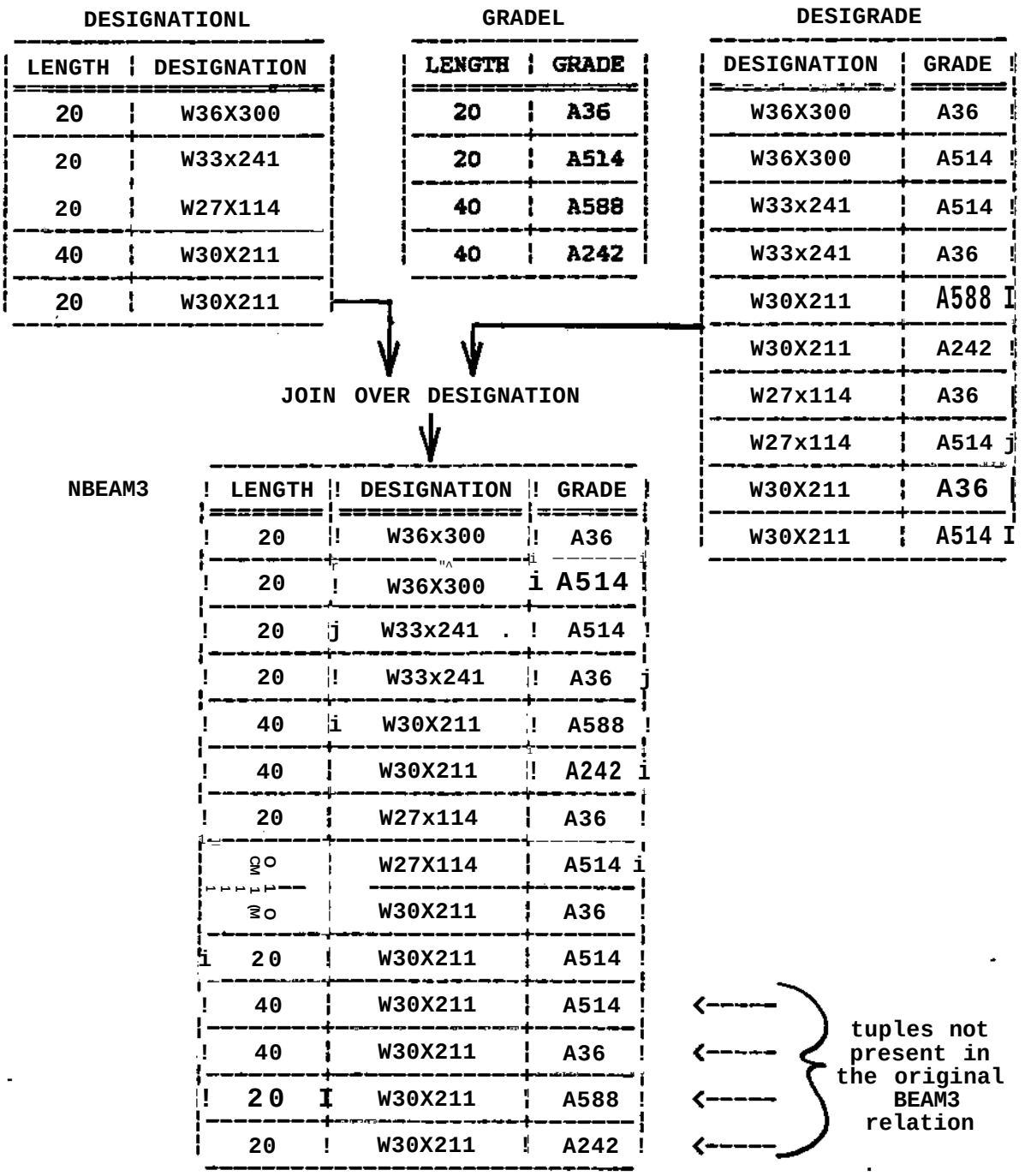

Figure 2-10: DESIGNATIONL, GRADEU DESIGRADE and NBEAM3 relations: Example of join dependency.

A relation is in fifth normal form or projection-join normal form if and only if all join dependencies are eliminated.

2.4.6. DOMAIN-KEY NORMAL FORM

After reading about all the other normal forms, one may wonder if there is a limit to the number of possible normal forms. It is true that as long as new problems are found, new normal forms could be defined. However, Fagin [11] has defined a different normal form that encompasses all of the other normal forms described in this chapter. It is called *domain-key* normal form (DK/NF). In order for a relation to be in DK/NF all *insertion* and *deletion* anomalies must be removed. Insertion and deletion anomalies are defined by Fagin [11] in the following way:

> "An insertion anomaly occurs when a seemingly legal insertion of a single tuple into a valid instance of the relation schema causes the resulting relation to violate one of the constraints of the relation schema. Here 'seemingly legal' means that every entry of the new tuple is in the appropriate domain and that the new tuple differs from all previous tuples".
>
> "A deletion anomaly occurs when the deletion of a single tuple from a valid instance of the relation schema causes a constraint to be violated".

All of the problems described in this chapter for the other normal forms can be described as an insertion and/or deletion anomaly, once the concept of *domain dependencies* is included in their definitions. A domain dependency is the dependency between the attribute and the domain. If the domain is infinite, there is no domain dependency. A bounded domain creates a domain dependency. For example, assume that the LENGTH domain in the BEAM3 relation in Figure 2-8 is bounded by the values 0 and 100. The insertion of a beam of length 200 feet would violate the domain dependency. Therefore, this is a special case of a insertion anomaly (a constraint is violated upon insertion).

Therefore, removal of these anomalies, by whatever means necessary, causes the relation to be in DK/NF which implies that the relation is also in first, second, third, fourth and fifth normal forms. The reader is directed to Fagin [11] for a proof of this statement and a more detailed description of DK/NF.

2.5. CONCLUSION

This chapter has defined some of the basic concepts in the relational approach to database management, using structural engineering examples. It has shown that databases, especially relational databases, are of great potential benefit to the engineering profession, due to increased data independence. Also, a detailed description of normalization has been provided. Normalization can best be summarized by the following statement:

> Once the user decides on the configuration (schema) of the database, he or she must also be sure that this schema guards against all possible insertion and deletion anomalies (problems) that might arise. If the schema does not do this, the user can choose one of the following alternatives:
>
> - Change the schema so that all possible problems are eliminated. If this is done, the database is in the "best" configuration.
>
> - Use this schema and manually check for the occurrence of the insertion and/or deletion problems. Under normal circumstances this alternative should not be chosen, since the insertion and deletion problems are usually very difficult to detect.

It is hoped that this chapter has provided the reader with an informative summary of database issues and has shown the importance of careful planning during the design of a relational database for engineering.

CHAPTER THREE
CONSTRAINTS

This chapter provides a detailed discussion of the use of design constraints in a relational DBMS, including the identification of the various sources and types of constraints. In addition, it contains a comparison between constraint enforcement in a normalized database and in a proposed database model containing relations augmented by constraint status attributes [19]. Extensions to this model are presented in Section 3.4.4.

3.1. USE OF CONSTRAINTS

As stated in Section 2.2, constraints are used to control the integrity of the database. They can be used: 1) to set minimum and/or maximum limits on the attribute values, and: 2) to insure that dependencies between attribute values are correctly maintained. The first class of constraints is supported by most DBMS's. The second class, however, is much more complex; the enforcement of the constraint could involve the retrieval of data from different attributes, tuples and/or relations. Therefore, this class of constraints is not fully implemented in current DBMS's. The discussion will concentrate on the second class of constraints.

3.2. CONSTRAINT SOURCES

Constraints arise from the following three main sources:

1. Physical or geometric relationships.
2. Codes and standards.
3. Design objectives or designer "style".

The following sections will describe and provide examples of each of these constraint sources.

3.2.1. PHYSICAL OR GEOMETRIC RELATIONSHIPS

Constraints which arise due to physical or geometric relationships are the easiest to understand. The constraint is usually based on the definition of one or more of the data items. An example of this kind of constraint (as shown in Section 2.2) is one that insures that the area of a solid rectangular beam is always equal to the height times the width:

$$\text{area} - \text{height} * \text{width} = 0$$

Another constraint of this kind could be based on purely geometric relations. Suppose a relation includes information about beams, such as beam ID number, length, yield stress (F_y), etc. Another relation contains information about sections of all the beams in the first relation, such as beam ID number, section ID number, section length, flange and web dimensions, etc. With these two relations in the database, a dependency arises between the length of each beam and the sum of the section lengths for that beam:

For each beam,

$$(\text{length} - \text{,2L,(section lengths)}) = 0.0$$

3.Z2. CODES AND STANDARDS

Design codes and standards provide many rules and guidelines governing acceptability of a specific design [15]. These rules can be modeled in a database as constraints. A typical example is the provision in the AISC Specification [2] that the actual bending stress (f_b) in a beam must be less than or equal to the allowable bending stress (F_u) for that beam. This constraint is shown below:

$$f_b \leq F_b$$

An entire specification can be represented as a set of such constraints [13], [14], [16], [20].

3.2.3. DESIGN OBJECTIVES OR DESIGNER "STYLE"

Individual designers can also impose constraints upon their particular design procedure in order to meet specific design objectives. For example. Section 1.10.2 (plate girders) of the AISC Specification [2] imposes the following constraint on the ratio of the clear distance between flanges to web thickness **(h/tj:**

$$h/t_w \leq 14{,}000/(F_y (F_y + 16.5))^{1/2} \quad (3.1)$$

where Fy is the yield stress of the steel. However, if transverse stiffeners are provided the limiting ratio is

$$h/t_w \leq 2000/(F_y)^{1/2} \quad (3.2)$$

Therefore, if transverse stiffeners are included in the design, constraint (3.2) must be checked, and constraint (3.1) does not have to be included. However, if the designer decides not to use transverse stiffeners, then (3.1) is the appropriate constraint. Thus, the specific constraint used is chosen by the designer to meet a particular design objective.

Other constraints, which are completely independent of a design code, can be imposed by a designer. A designer may decide that only one grade of steel will be used and therefore specify a constraint which limits the value of the yield stress to the value which corresponds to the specific grade. Constraints could also be specified to limit beam dimensions, stresses or deflections to values within an acceptable range determined by the designer based on previous experience.

3.3. CONSTRAINT TYPES

The type of a constraint may be determined by the location of the data items needed to evaluate the constraint. For example, if all the data necessary to evaluate a particular constraint is stored in one single tuple of a relation, then the constraint is considered a *single relation-single tuple* constraint. Since some constraint types are more complex to evaluate than others, due to the effort spent on retrieving the necessary data items, it is important for the database schema designer to be aware of the types that would have to be included in any proposed schema. Therefore, this section will describe and provide examples of the various constraint types.

Section 3.3.1 will discuss constraint types that insure integrity within a single relation. In Section 3.3.2, the discussion is expanded to include constraints that insure integrity between data in two or more relations.

The examples used in this section are based on the following two relations:

BEAMS(6eamid, blend, brend, blength)

SECTIONS(£eam/(/, *sectionid,* slend, srend, slength, width, height, area)

The relation name is listed first, followed by the names of all of its attributes. The attributes in italics are the key for the relation. The BEAMS relation contains the beam identification number (beamid - key), left end location with respect to a datum, right end location and the length of each beam. The SECTIONS relation contains the beam and section identification numbers (beamid, sectionid - composite key), left end location with respect to the left end of the beam, right end location, segment length, width, height and area of each solid rectangular section of each beam in the BEAMS relation.

3.3.1. SINGLE RELATION CONSTRAINTS

As stated above, single relation constraints insure integrity within a single relation. There are three main types of single relation constraints:

1. Single relation-single attribute.
2. Single relation-single tuple.
3. Single relation-multiple tuple.

Single relation-single attribute constraints have already been defined. They restrict the allowable range for a specific attribute. For ~ example, the following single relation-single attribute constraints could be imposed:

BEAMS.blength > 0.0

BEAMS.blength £ 100.0

SECTIONS.slength £ 0.0

SECTIONS.slength £ 100.0

SECTIONS-width > 0.0

SECTIONS.height > 0.0

SECTIONS.area > 0.0

Notice that each expression constrains only a specific attribute value in a single tuple of a single relation. This simple type of constraint is implemented in essentially all DBMS's. The constraints limiting the beam lengths can be specified as a SEQUEL 2 [6] assertion (discussed in Section 2.3.2.4) in the following manner

ASSERT A1 **ON** BEAMS: blength **BETWEEN** 0.0 **AND** 100.0

Single relation-single tuple constraints are used to enforce dependencies between attributes in each tuple of a single relation. The dependencies can involve simple comparisons between attributes in a specific tuple, such as:

SECTIONS.slend £ SECTIONS.srend

This constraint forces the value for the left end location of a section to be less than the right end value for each section in the SECTIONS relation, i.e. positive distance is measured from left to right.

This type of constraint can also require mathematical operations:

SECTIONS-width X SECTIONS.height - SECTIONS.area = 0.0

BEAMS.brend - BEAMS.blend - BEAMS.blength = 0.0

SECTIONS.srend - SECTIONS.slend - SECTIONS.slength = 0.0

The first constraint requires the area of a particular section to be equal to its width times its height. The next two constraints require beam and section lengths to equal the distance between their right and left ends. In SEQUEL 2, the first constraint between the right and left end locations is specified as follows.

ASSERT A2 **ON** SECTIONS: slend ≤ srend

The single relation-single tuple constraints are also fairly simple to implement, since they require data retrieval from only one tuple in a specific relation each time the constraint is checked. However, the specification of mathematical operators in the constraint introduces some amount of complexity. Therefore, some DBMS's allow only a restricted form of the single relation-single tuple type constraints with only relational operations (=, #, >, £, <, £) permitted in the constraint specification.

Single relation-multiple tuple constraints are much more complex than the two types discussed above. This type is used to insure dependencies between data items in different tuples of a specific relation. An example of this type of constraint is listed below. This constraint insures the design objective that the height difference between two adjacent sections is less than a specified constant.

| SECTIONS.heightX - SECTIONS.heightY| £ CONSTANT

WHERE SECTIONS.beamidX = SECTIONS.beamidY **AND**
SECTIONS.slendX = SECTIONS-srendY

The X and Y following the attribute name indicate different tuples, i.e. Xth tuple and Yth tuple. Also, the | | indicates absolute value.

To check the constraint the relation is searched to select a group of sections that have a common beam identification number. Then all the srend and slend attributes in this group are compared to find adjacent sections so that the height attributes from the two sections can be checked. Therefore, data from two tuples is needed every time this constraint is checked. Since most DBMS's do not allow constraints that require comparisons between two or more tuples, this type of constraint cannot be enforced. However, SEOUEL 2 does allow these type constraints[3], since its assertion component includes the ability to select (query) the appropriate data and then check the constraint.

ASSERT A3 ON SECTIONS : | X,height - Y,height| £ constant

WHERE X,beamid = Y,beamid **AND** X,slend = Y,srend

3.3.2. MULTIPLE RELATION CONSTRAINTS

Multiple relation constraints are used to insure integrity between data items in different relations. There are three main types:

1. Multiple relation-single attribute
2. Multiple relation-single tuple.
3. Multiple relation-multiple tuple

---

3 SEQUEL 2 excludes the use of mathematical operations within the constraints, but this restriction will be ignored in the discussion.

Multiple relation-single attribute constraints enforce dependencies between single attribute values in two or more relations. The only difference between this type of constraint and the single relation-single attribute constraints is that the attribute value is compared to another attribute value in a different relation instead of being compared to a user supplied constant. For example, to insure that each section in the SECTIONS relation has a beamid equal to one of the beamid's in the BEAMS relation, the following constraint is specified:

SECTIONS.beamid = BEAMS.beamid

This constraint arises due to the hierarchical relationship between attributes, since the information in the SECTIONS relation corresponds to specific sections of the beams in the BEAMS relation; a section cannot exist if the beam itself does not exist.

This type of constraint is more difficult to enforce than the single relation types, since it involves retrieval from two or more relations. However, the following SEQUEL 2 assertion can be specified to enforce the example constraint:

```
ASSERT A4:

    (SELECT beamid FROM SECTIONS)
    IS IN
    (SELECT beamid FROM BEAMS)
```

Multiple relation-single tuple constraints insure dependencies between individual tuples in two different relations. The only difference between this constraint type and the previous one is that more than one attribute from each tuple is needed to evaluate the constraint. For example, the following multiple relation-single tuple constraint is specified to force an individual section length to be less than or equal to its corresponding beam length (a section cannot be longer than the beam):

SECTIONS.slength ≤ BEAMS.blength

WHERE BEAMS.beamid = SECTIONS.beamid

To evaluate this constraint, data is required from two attributes in single tuples of two different relations. The constraint is specified in SEQUEL 2 in the following manner:

ASSERT A5: **ON** SECTIONS X: slength <=

(SELECT btength
FROM BEAMS
WHERE beamid = X,beamid)

Multiple reiation-multiple tuple constraints insure dependencies between data in two or more tuples in two or more relations. These constraints usually require *aggregate data,* such as averages, summations, minimum and/or maximum values of particular attributes in all tuples of specific relations. Since, as shown by [3], the aggregate information can be "designed to be quickly accessed and easily maintained", multiple relation-multiple tuple constraints which require aggregate data may not be as complex as other multiple relation-multiple tuples constraints. However, it is assumed that the more complex constraints are still enforceable, given access to all necessary retrieval operations. The following is an example of a multiple relation-multiple tuple type constraint which requires aggregate data:

BEAMS.blength - ^jSECTIONS.slength) = 0.0

WHERE BEAMS.beamid = SECTIONS.beamid

where $\sum_{x}$ indicates a summation operator. This constraint forces the beam length in BEAMS to equal the summation of all the lengths of all sections of each beam. It can be specified in SEQUEL 2 in the following manner

ASSERT A6 **ON** BEAMS X: blength =

(SELECT SUM(slength)
FROM SECTIONS
WHERE beamid = X,beamid)

An example of a multiple relation-multiple tuple constraint that does not require aggregate data is one which insures the design objective that the height difference between two adjacent sections is less than one tenth of the overall beam length.

| SECTIONS-heightX - SECTIONS.heightY| £ BEAMS.blength/10

WHERE BEAMS.beamid = SECTIONS.beamidX **AND**
SECTIONS.beamidX = SECTIONS.beamidY **AND**
SECTIONS.slendX = SECTIONS.srendY

It can be specified in SEQUEL 2 in the following manner

```
ASSERT A7 ON SECTIONS : (|X,height - Y,height| ≤
    (SELECT blength
     FROM BEAMS
     WHERE beamid = X,beamid)/10)

     WHERE X,beamid = Y,beamid  AND X,slend = Y,srend
```

3.4. CONSTRAINT ENFORCEMENT ALTERNATIVES

This section will present three alternatives for constraint enforcement in an engineering design database.

3.4.1. NORMALIZED DATABASE

Normalization has been the most common method used to remove dependencies between data items, by projecting out the dependent or "redundant" attributes [15]. Using the remaining independent attribute values, the user or application program can then determine the values of the deleted redundant attributes.

As stated in [15], one objection to normalization by deleting the dependent attribute(s) is that it is usually difficult to determine which attributes are dependent or independent. Also, during the design process, attributes can change from independent to dependent, and vice versa.

An example of this (similar to the one presented in [15]), would be the dependence between the width, height and area of solid rectangular beams. At one point the designer may determine the necessary beam area to satisfy a specific stress constraint and then select the beam dimensions which provide that area. At some other time, the designer may determine the beam dimensions first from clearance constraints, and then calculate the resulting area. Therefore, at specific points of the design process, the designer would need to delete different attributes from the database to eliminate the attribute dependencies. One of the following expressions would be used to calculate the value of the dependent attribute (depending upon which attribute was chosen):

SECTIONS.area = SECTIONS.width X SECTIONS.height

SECTIONS.height = SECTIONS.area / SECTIONS.width

SECTIONS.width = SECTIONS.area / SECTIONS.height

A second objection to normalization is that although dependencies can be removed so that equality constraints can be enforced, a constraint which specifies an inequality between data items cannot be enforced. For example, a constraint could specify that a section length must be greater than or equal to twice its height. To eliminate this dependency, the length attribute would have to be deleted; however, this would cause the loss of independent data, since the actual length values probably depend upon other factors and cannot be calculated by just doubling the height value.

Finally, a third objection to normalization is that it does very little (if anything at all) for multiple relation constraints. For example, the two relations used in the previous section can be normalized with respect to the length, the left and the right end locations by eliminating the brend and srend attributes, as shown below.

BEAMSl&ea/n/tf, blend, blength)

SECTI0NS(£ea/7?/i/, *sect/on id,* slend, slength, width, height, area)

Within a particular relation, none of the remaining information concerning the length or location of the beams or sections can be eliminated without losing independent data. However, the dependency between the beam length and the summation of section lengths still exists.

Thus, even though normalization eliminates some dependencies within relations, it does not provide the necessary flexibility for the design process. Therefore, an alternative is needed that will insure all dependencies are satisfied and also provide flexibility.

3.4.2. **DIRECT CONSTRAINT ENFORCEMENT**

An alternative to normalization for constraint enforcement is to add integrity rules to the DBMS so that it can enforce all of the possible constraint types illustrated in Section 3.3. Therefore, if an insertion or update causes a constraint to be violated, the specific action is immediately rejected by the database system and no change is made. This alternative is capable of insuring all dependencies, however, it is still not flexible enough for design purposes, since the database requires the constraints to be satisfied at all times. During preliminary design, certain constraints are ignored to allow for the testing of trial solutions or because some data items have not yet been calculated. Therefore, a database system that provides this additional constraint control must be developed.

3.4.3. AUGMENTED DATABASE, SINGLE RELATIONS

This section summarizes a database system, proposed by Rasdorf [19], [15], which contains relations augmented by *constraint status attributes.* A constraint status attribute is added for each constraint to record the status of the constraint and can have a value of either TRUE (for constraint satisfied) or FALSE (for constraint violated). This extra attribute is added to the specific relation that is constrained by the particular constraint. For example, to check the constraint on the area, width and height of a section, the SECTIONS relation is modified as follows:

SECTIONS(£ea/77/'tf, *sect ion id,* srend, slend, slength, width, height,

area, areaok)

where the areaok attribute is the status attribute for the constraint. Section 3.4.4 will discuss the proper placement of these attributes for constraints which involve more than one relation.

3.4.3.1. CONSTRAINT EVALUATION

The constraint status is determined by the database using either user supplied checking functions which can evaluate the status attribute by checking the constraint or assignment procedures which can assign a value to the

dependent data item such that the constraint is satisfied. For example, the status of the area constraint can be evaluated by using either the PASCAL function or the PASCAL procedure show below.

```
FUNCTION AREAOK(area, width, height:REAL): BOOLEAN
    BEGIN
        AREAOK := (ABS(area - width*height) <= .01)
    END


PROCEDURE SETAREA(width, height:REAU VAR arearREAL;
                  VAR areaok:BOOLEAN)
    BEGIN
        area := heigbt«width
        areaok := TRUE
    END
```

Notice that the SETAREA procedure calculates the value for the area. Similar procedures can be specified to calculate the height or width. Section 3.4.3.2 will show how the user can specify which assignment procedure to use. These routines assume that all necessary data is available and can be passed to them. Rasdorf's presentation [19], only discussed constraint routines whose arguments are attribute values in a single tuple of a single relation. In general this is not always the case. Section 3.4.4 will extend this discussion to multiple tuple and multiple relation constraints.

3.4.3.2. CONSTRAINT CONTROL

As stated in Section 3.4.2, the database must permit the user to control which constraints are to be enforced at any given time. The database system proposed in [19], [15] provides this by including a control mechanism which supports the following three control commands:

- INVOKE(constraint)
- ACTIVATE(constraint)
- DEACTIVATE(constraint)

where the word constraint is replaced by the name of the constraint checking function or assignment procedure.

The INVOKE command causes the database to call the specified constraint routine to check the constraint for each tuple currently in the database. If the

procedure encounters any undefined, uninitialized or "null" values while selecting data, the constraint status attribute is automatically set to false. This allows the user to continue with the design process even though pieces of data are missing or constraints are violated. At some other point in the design process, the user can **SELECT** the status attributes that were set to FALSE to correct or load the appropriate pieces of data. The INVOKE command also can be combined with a where clause so that the constraint is invoked on only a specific set of tuples.

The ACTIVATE command causes the database to call the specified routine for each insertion or update of the database, and accept the change only if the constraint status attribute evaluates to TRUE.

The DEACTIVATE command causes the database to ignore the constraint so that changes can be made without checking. When the changes are completed, the constraint can be invoked and activated again.

As stated in [15], complete integrity "can be assured only if ACTIVATE is preceded by INVOKE". Therefore, in this study, it is assumed that the ACTIVATE command automatically performs the INVOKE, warns the user of any non-conforming data and then activates the constraint. Since the INVOKE command can be combined with a where clause, the specification of the ACTIVATE command can include a where clause which is used for the INVOKE only.

3.4.4. AUGMENTED DATABASE, MULTIPLE RELATIONS

The following will describe the necessary extensions to the database system described in the previous section which eliminate specific problems with the model as well as provide the enforcement of multiple tuple and multiple relation constraints. Two extensions to the constraint control mechanism are needed. First, there must be a way to determine which constraints are activate at any specific time. For example, a user could insert new values into the database and invoke and activate the constraints so that they are all satisfied. Then another user could deactivate a particular constraint and change the values of some of the data such that the deactivated constraint may or may not be satisfied. If this data is then retrieved, it cannot be

assumed to be valid unless all constraints that depend on this data are reactivated before the retrieval. This would not be very efficient since most of the constraints have not been deactivated. However, if the database system kept a record of all of the activated constraints, the user could specify which constraints are to be satisfied, and then the database would only activate the constraints that were not currently activated.

Second, although the current mechanism gives the user control over the enforcement of constraints, there is a definite problem in the way the constraints are checked. As stated above, once a constraint is activated, the database permits changes only if the changes satisfy the constraint. Therefore, for an equality constraint whose status attribute currently has a value of TRUE, no changes will ever be allowed, since the constraint is checked after every single change. For example, suppose the areaok attribute is equal to TRUE for a specific beam, however, a user intends to change the width of this beam. To do this, the user would also have to change the area so that the constraint is again satisfied. This process cannot be accomplished if the constraint is currently activated; the first update (on width) violates the constraint, so it is rejected. The second update (on area) also violates the constraint and is also rejected. The update can only be accomplished by deactivating the constraint, specifying the changes and then invoking the constraint, in which case, the ACTIVATE command is of no use.

This problem can be eliminated if *multiple updates* are allowed. A multiple update enables a user to complete an entire transaction (containing many insertions or updates) before any constraints are checked. Once the transaction is completed, the appropriate constraints are checked and the transaction is either accepted or rejected, in which case the entire transaction must be "backed out" to restore the database to its original state before the transaction began.

In SEQUEL 2 a user can specify a multiple update by placing the update commands "between the statements BEGIN TRANSACTION and END TRANSACTION" [6]. All assertion checking is ignored until the entire transaction is processed (unless the word IMMEDIATE was included in the assertion specification, in which case the assertion is always checked).

The need for multiple updates also arises in the enforcement of multiple relation constraints. Such an example is the one which forces each beam length to equal the summation of each of its section lengths. Since this constraint involves multiple tuples and multiple relations, it cannot be enforced in the database system described in the previous section. However, if the checking and assignment routines in that system are given access to all of the database query operations, this type of constraint can be enforced [15]. Therefore, the model must be extended to include the ability of specifying database query operations in all constraint procedures and functions.

Enforcement of this constraint requires the addition of one constraint status attribute to the BEAMS relation to record the status of the constraint. The status attribute is added to the BEAMS relation because this constraint is checked for each beam. If it were added to the SECTIONS relation, a large amount of redundant data would be introduced to the database since, for each section of a specific beam, the status attribute has the same value.

With the new attribute (lengthok) added, the example database has the following structure:

BEAMS(beamid, blend, brend, blength, lengthok)

SECTIONS(£ea/n/<*/, sect ion/d,* slend, srend, slength, width, height,

area)

In order to evaluate the status attribute, the following PASCAL-like function is used:

```
FUNCTION LENGTHOK(beamnum:char): BOOLEAN
    BEGIN
        DBMS("SELECT(blength into blen from BEAMS where beamid =
                    beamnum,
                    SUM(slength) into sumlen, COUNT into num
                    from SECTIONS where beamid = beamnum,
                    selectok)")
        IF (selectok) AND (num => 1) THEN
            lengthok := (ABS(blen - sumlen) <= .01)
        ELSE
            lengthok := 'false'
            ERRORC'Error or null value in BEAMS and/or
                    SECTIONS relation")
    END
```

The DBMS procedure is called to perform the query operations. The argument of this procedure specifies the query operations to be executed. In this study, the DBMS procedure calls will use a query language similar to SEQUEL 2. Also, the "BEGIN and END TRANSACTION" commands from SEQUEL 2 are implied by the DBMS procedure. For example, in the above example, it is assumed that the two **SELECT** operations are one complete transaction. The SELECT procedure is used to select the length attribute (blength) from BEAMS for the current beam (beamnum) and the summation of the section lengths (slength) and the number of sections selected (num) from SECTIONS for all sections of the current beam (beamnum). The selectok variable is used to "flag" the success or failure of the select operations. If the selectok variable is FALSE or if the number of sections selected (num) is zero, the constraint cannot be checked and the function assumes that there are null values or an error in one or more of the relations. Therefore, the constraint status attribute (lengthok) is set to FALSE. Notice that if no sections are selected, the value of selectok is TRUE. This is because selecting a COUNT equal to zero is considered a successful select operation.

When a user tries to update a beam's length and/or any of its section lengths, the database system can call this function and allow or reject the update depending upon the value of the constraint (lengthok). Again, this assumes that multiple updates are allowed. If the user can not specify a multiple update, the database need only check the value of lengthok. If lengthok has the value FALSE, the database calls the function and acts on the update accordingly. If lengthok has the value of TRUE, then the update is automatically rejected without calling the function (since changing only a beam length or one section length will violate the constraint). This eliminates some processing effort; however, it also removes some user flexibility. If the user's complete update would satisfy the constraint, it is still rejected, since the database looks at each specific change.

A second example of the proposed constraint enforcement mechanism is shown below. This example enforces the constraint which insures the design objective that the height difference between two adjacent sections is less than a specified constant.

FUNCTION HEIGHTOK(beamnum, sectnumxhar): BOOLEAN

```
BEGIN
  DBMS("SELECT(height into hf srend into rendr slend into
                lend from SECTIONS where beamid = beamnum
                and sectionid = sectnum,
                height into height1, COUNT into numi from
                SECTIONS where beamid = beamnum and slend =
                rend,
                height into height2, COUNT into num2 from
                SECTIONS where beamid = beamnum and srend =
                lend,
                brend, blend from BEAMS where beamid =
                beamnum, selectok)")
  IF (selectok) THEN
    BEGIN
      TF (numi = 0 AND rend = brend) THEN
          height1 := h
      IF (num2 = 0 AND lend = blend) THEN
          height2 := h

      heightok := (abs(h - height!) £ constant)
                   AND (abs(h - height2) ≤ constant)
    END
  ELSE
    BEGIN
      heightok := 'false'
      ERRORC'error or null value in  BEAMS or SECTIONS")
    END
END
```

If the select operations are unsuccessful, the function assumes that there are null values or an error in the BEAMS or SECTIONS relations. If the number selected from the second or third selects equals zero, and the current section is at one of the beam ends, then the function automatically sets the value of height so that the constraint is satisfied for that end of the section.

The proposed constraint enforcement alternative seems to provide the designer with a large degree of flexibility by allowing the enforcement of all constraint types and by giving the user complete control over which constraints are to be enforced at any particular time. The next chapter will describe a design example that will be used in Chapter 5 to compare the use of this alternative to the normalized database alternative.

CHAPTER FOUR
DESIGN EXAMPLE

This chapter describes a specific design example used in Chapter 5 to compare two of the constraint processing alternatives discussed in Section 3.4, namely, normalization versus augmented databases.

Included in this description is the definition of the data items in the database, the database schema, and the constraints that must be enforced by the user or program in the first alternative or by the database in the second alternative.

4.1. DESIGN PROCEDURE

The example deals with the design of the girders of a continuous plate girder bridge. It involves the following three main design phases:

- Conceptual Design
- Beam Sizing
- Splice and Stiffener Design

4.1.1. CONCEPTUAL DESIGN

The conceptual design phase involves the selection of the preliminary beam dimensions using a few specific design constraints, such as basic constraints for allowable stress or allowable dimensions. This phase receives as input all of the basic information about the overall structure, layout and assumed behavior, such as segment lengths, locations and moment type (design for positive or negative moment segments), since this information is based on the designer's experience. This information must be checked before any calculations can be performed. Therefore, all appropriate constraints are checked at the beginning of this phase.

4.1.2. BEAM SIZING

The beam sizing phase uses the results of an analysis procedure to select the final beam dimensions. Three steps are involved: first check the corresponding constraints for all input data; second call the analysis procedure to perform the static analysis; and third, check the design constraints using the preliminary dimensions and analysis results. If any constraints are violated, the necessary changes are made, the analysis is repeated with the new values and the design constraints are rechecked. This process is repeated until all constraints are satisfied.

4.1.3. SPLICE AND STIFFENER DESIGN

In the splice and stiffener design phase, the design of all bearing stiffeners and beam splices is performed. The process includes checking all appropriate constraints on the input data, the selection of all dimensions for the stiffeners, splice plates and bolts, and checking the corresponding design constraints to insure that these dimensions are satisfactory.

4.2. DATA ITEMS

Figure 4-1 contains a list of each data item stored in the database along with its corresponding definition.

The entire database schema, including the relations and the single relation-single attribute constraints checked by the database is shown in Figure 4-2. The STRUCTURE relation contains information that remains constant throughout the design procedure, such as allowable stresses and deflections, overall girder length and grade of steel of the structure. The SEGMENTS relation contains information about the length and location of each segment of the girders. This relation identifies the points of the girder in which the section flange dimensions change. The ends of each section are usually located at a point of minimum moment. Since it is assumed that only two different flange sections are designed (one for positive moment sections, one for negative moment sections), the posmom attribute is used to differentiate between the two sections. The WSECTIONS relation contains the design alternative number and the web dimensions for the entire girder. The FSECTIONS relation

ALTERNATIVE	Identification number for alternative designs of the current structure.
ANALOC	Analysis result point measured from the left end of section (expressed as a function of the section length). For example, a value of .2 specifies a distance of two tenths of the section length from the left end.
BBS	Width of the bearing stiffener.
BF	Flange width of a particular section.
BI	Width of the inside flange splice plate.
BO	Width of the outside flange splice plate.
BOLTDIA	Bolt diameter to use for all connections.
BWS	Width of the web splice plates.
BSLOC	The location of the bearing stiffeners measured from the left end of the girder.
CLEAR	Allowable vertical clearance for the girder.
DEFALL	Allowable deflection limit (expressed as the denominator of the length to deflection ratio. For example, if the deflection must be less than the span length divided by 1000 (1/1000), the value of DEFALL would be 1000.
DEFY	Calculated vertical deflection at a particular analysis point of a section.
E	Modulus of elasticity of steel.
FBALL	Allowable bending stress in the structure.

Figure 4-1: Data Items for Design Example.

FBBOLT	Allowable shear force in a bolt.
FNBOLTS	The total number of bolts needed for the flange splice.
FNLINES	The number of lines of bolts on each side of the flange splice.
FSLENGTH	Total length of the flange splice plates.
FSLOC	Location of the center line of the flange splice measured from the left end of the girder.
FV	Allowable shear stress in the structure.
FY	Yield stress of steel.
GRADE	Grade of steel used in the structure.
H	Web height of a particular section.
HBS	Height of the bearing stiffener.
HOLEDIA	Diameter of all drilled bolt holes.
IX	Moment of inertia of a particular section.
LENGTH	Length of the girder.
LOAD	Character variable for loading type. For example, load equals 'dload' for the analysis results corresponding to the analysis performed using only the dead load.
MOM	Calculated moment at a particular analysis point of a section.
NUMGIRDER	The number of girders used across the width of the bridge.

Figure 4-1: Data Items for Design Example, continued.

POSMOM	Logical variable; TRUE if current section is to be designed for positive moment; FALSE if current section is to be designed for negative moment.
ROT	Calculated rotation at a particular analysis point of a section.
SECTIONID	Identification number for each section.
SHEAR	Calculated shear force at a particular analysis point of a section.
SLEND	Location of the left end of a section with respect to the left end of the beam (assumed to be a distance of 0).
SLENGTH	Length of a section.
SUPPORTLOC	The location of the supports measured from the left end of the girder.
TBS	Thickness of the bearing stiffener.
TF	Flange thickness of a particular section.
TI	Thickness of the inside flange splice plate.
TO	Thickness of the outside flange plate.
TW	Web thickness of particular section.
TWS	Thickness of the web splice plate.
WNBOLTS	The total number of bolts needed for the web splice.
WNLINES	The number of lines of bolts on each side of the web splice.
WSLENGTH	The total length of the web splice plates.
WSLOC	The location of the center line of the web splice measured from the left end of the girder.

Figure 4-1: Data Items for Design Example, continued.

RELATIONS

STRUCTURE(grade, fball, fv, e, detail, length, clear, boltdia, holedia. fbbolt)

SEGMENTS(a/te/77af/Ve. *sectionid,* slength, slend, posmom)

WSECTIONS<a/te/77af/ve, h, tw)

FSECTI0NS(a/te/7?af/Ve, *posmom,* bf, tf)

ANA\yS\S(alternative. sectionid, load, analoc, shear, mom, defy, rot, ix)

G\ROER(alternative, numgirder)

SUPPORTS(a/ter/?af/Ve, supportloc)

BSTIFFENERS(a/ter/7af/Ve, *ósioc,* hbs, tbs, bbs)

FSPLICES(a/ter/?af/Ve, *fsloc,* bo, to, bi, ti, fslength, fnlines, fnbolts)

WSPUCESia/ternative, wsfoc, bws, tws, wslength, wnlines, wnbolts)

GRADESigrade, fy)

SINGLE RELATION-SINGLE ATTRIBUTE CONSTRAINTS

0 < fball £ MAXfball	0 < clear ≤ MAXclear
0 < fv £ MAXfv	0 < boltdia ≤ MAXboltdia
0 < e £ MAXe	0 < holedia £ MAXholedia
0 < defall £ MAXdefall	0 < fbbolt £ MAXfbbolt
0 < length £ MAXlength	0 < slength £ MAXslength

Figure 4-2: Database schema for normalized design database example.

0 < slend £ MAXslend

0 < tw £ MAXtw

0 < tf £ MAXtf

0 < numgirder £ MAXnumgirder

0 < bsloc £ MAXbsloc

0 < tbs £ MAXtbs

0 < fsloc ≤• MAXfsloc

0 < to £ MAXto

0 < ti £ MAXti

0 < fnlines £ MAXfnlines

0 < wsloc £ MAXwsloc

0 < tws £ MAXtws

0 < wnlines £ MAXwnlines

0 < fy £ MAXfy

0 < h £ MAXh

0 < bf £ MAXbf

0 < anaioc £ 1.0

0 < supportloc £ MAXsupportloc

0 < hbs £ MAXhbs

0 < bbs £ MAXbbs

0 < bo ≤: MAXbo

0 < bi ≤. MAXbi

0 < fslength £ MAXfslength

0 < fnbolts £ MAXfnbolts

0 < bws £ MAXbws

0 < wslength £ MAXwslength

0 < wnbolts £ MAXwnbolts

Figure 4-2: Database schema for normalized design database example, com.

contains the alternative number and the flange dimensions for the positive and negative moment sections of the girder. The ANALYSIS relation contains all of the results of a static analysis procedure applied to the structure. The GIRDER relation stores the number of girders in the specific design alternative. The SUPPORTS relation contains the support locations for each design alternative. The BSTIFFENERS, FSPLICES and WSPLICES relations contain the information (alternative, location and dimensions) describing the bearing stiffeners, the flange splices and the web splices, respectively. Finally, the GRADES relation contains the grade and yield stress for different grades of steel.

Input and output data for each design phase is contained in two specific sets of relations. Figure 4-3 shows the relations used for input and output for the three phases.

4.3. CONSTRAINTS

All of the constraints needed for the design example are listed in Figure 4-4, including their definitions and sources. These constraints use data items retrieved from the database as well as local variables calculated from the retrieved data. The source of some of the constraints is listed as "the design code". Since the design example is based on an actual bridge design, the particular design constraints used are taken from a state bridge design manual. Also, notice that some of the constraints use a function called TOLERANCE. This function returns a value to use as a tolerance in the particular constraint. Table 4-1 provides a list of each constraint and the design phases in which it is used.

The total design procedure uses additional constraints that are not listed in Figure 4-4. These constraints deal with overall decisions made about the design procedure before the design begins. For example, this design assumes that all connections are bolted. This assumption is not really necessary, since the program or database could include the constraints for all other types of connections and use only the ones that apply to the type of connection used. Inclusion of such additional constraints only adds complexity and does not add to the discussion of the constraint enforcement alternatives.

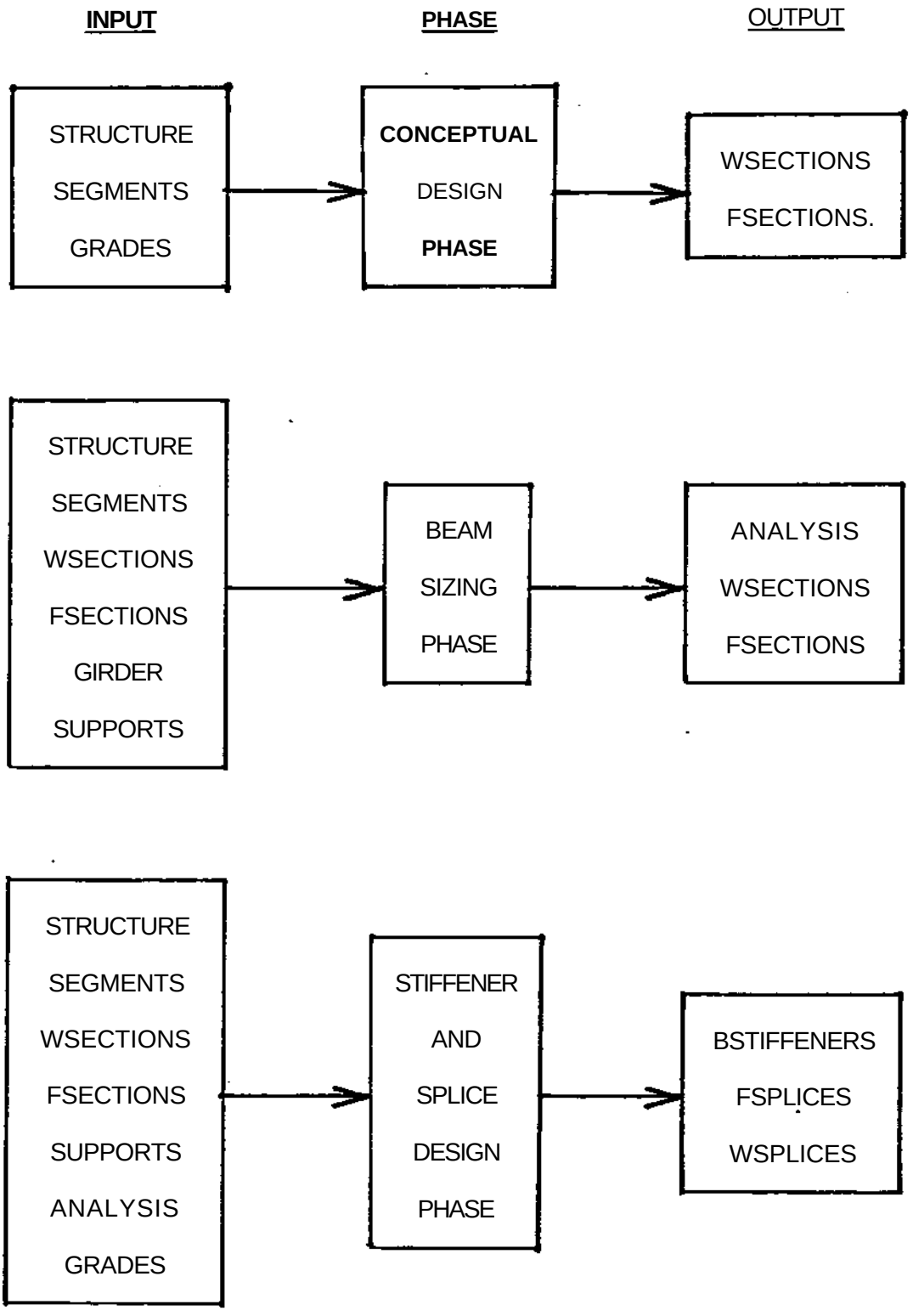

Figure 4-3: Data flow for the three design phases.

AFREMOK	afremok := .75 £ (bf - (fnbolts/fnlines)*holedia)/bf
	This constraint from the design code insures that at least 75% of the flange area remains after the bolt holes are drilled.
AWREMOK	awremok := .75 £ (h*tw - (wnbolts/wnlines*holedia*tw)/(h*tw)
	This constraint from the design code insures that at least 75% of the web area remains after the bolt holes are drilled.
BBSOK	bbsok := bbs £ (bf - tw)/2
	This physical constraint insures that the bearing stiffener is wider than the overhanging flange.
BEAMOK	beamok := htok AND stressok AND defok AND iok
	This design objective constraint insures that all constraints for the beam sizing phase are satisfied.
BEARINGOK	bearingok := reaction/(2*tbs*bbs) £ .9*fy
	This constraint from the design code insures that the force in the bearing stiffener does not exceed its bearing capacity. The variable "reaction" is the calculated reaction at the support.
BIOK	biok := (bi £ (bf - tw)/2 AND bi £ ((fbbolts/fnlines)/2 + 1)»2)
	This physical constraint insures that the width of the inside flange splice plate is no larger than the overhanging flange and large enough for the bolt holes.
BOLTFOK	boltfok := boltf £ fbbolt
	This constraint from the design code insures that the force on each bolt (boltf) in the web splice is no greater than the allowable force.
BOOK	book := (bo £ bf AND bo ≥ (tw • ((fnbolts/fnlines) + 2)*2))
	This physical constraint insures the the width of the outside flange splice plate is no greater than the flange width and large enough for the bolt holes.

Figure 4-4: Constraints used in the design example.

BSLOCOK — bslocok := (bsloc is equal to supportloc in SUPPORTS)

This design objective constraint insures that the bearing stiffener is located at a support.

BWSOK — bwsok := (bws ≤ h AND bws ≥ ((fbbolts/fnlines) - 1)*3.5 + 3)

This physical constraint insures that the width of the web splice plate is no greater than the web height and large enough for the bolt holes.

CCOK — ccok := cc ≥ k*hbs/r

This constraint from the design code insures that the column slenderness ratio (cc) for the bearing stiffener is acceptable. The variables "k" and "r" are the effective length and radius of gyration of the stiffener, respectively.

CHANGEOK — changeok := abs(tf2 - tf1) ≤ TOLERANCE('changeok')

This design objective constraint insures that the difference in height between two adjacent sections is less than a specified tolerance. Since the height of the web is constant throughout the girder, the constraint only has to compare flange thicknesses (tf1 and tf2).

CLEAROK — clearok := abs(h + tf*2 - clear) ≤ TOLERANCE('clearok')

This physical constraint insures that the total height of a girder (web height plus flange thicknesses) is no greater than the actual clearance.

CONCEPTOK — conceptok := conhtok AND coniok AND coniflangeok AND conflangeok

This design objective constraint insures that all of the constraint for the conceptual design phase are satisfied.

CONFLANGEOK — conflangeok := $bf/tf \leq 65/(fy)^{1/2}$

This constraint from the design code insures that the slenderness ratio of the flange does not exceed a specific maximum.

Figure 4-4: Constraints used in design example, continued.

CONHTOK

conhtok := $h*((shear*1000)/(h*tw))^{1/2}/7500 \leq tw$

This constraint from the design code was chosen to satisfy the design objective that no transverse stiffeners be included in the design. It restricts the web slenderness ratio to be less than a specific maximum. This constraint is exactly the same as HTOK except that it is used in the conceptual design phase before an analysis is performed and the exact value of shear is known.

CONIFLANGEOK

coniflangeok := $bf*tf*(h/2)^2*2 \geq$ iflanges

This physical constraint insures that the actual moment of inertia of the flanges is no less than the required moment of inertia (iflanges).

CONIOK

coniok := $ABS(h - (3*h*sreqd/(2*tw))^{1/3}) \leq$ TOLERANCE('coniok')

This physical constraint insures that the web height provides the required moment of inertia. The variable "sreqd" is the required section modulus.

CONNOK

connok := (abs(slend(1) + slength(1) - lend) ≤ TOLERANCE('connok') AND ABS(lend + seclength - slend(2)) ≤ TOLERANCE('connok'))

This physical constraint insures that the current section is connected to another section on each side. The variables "lend" and "seclength" are the left end and section length of the current section, respectively. In a complete implementation, a check is needed to verify if one or both ends of the section correspond to the beam end(s).

DEFOK

defok := l/defall ≥ defy

This constraint from the design code insures that the deflection in a particular section does not exceed the maximum allowable deflection of the current span. The variable "l" is the current span length.

FNBOLTSOK

fnboltsok := ABS(fnbolts - tencap/fbbolt) ≤ TOLERANCE('fnboltsok')

This constraint from the design code insures that a sufficient number of bolts are used for the flange splice. The variable "tencap" is the capacity of the flange.

Figure 4-4: Constraints used in design example, continued.

FORCEOK forceok := reaction £ p

This constraint from the design code insures that the actual force in the bearing stiffener (the reaction at that location) is not greater than the allowable force (p).

FSLENGTHOK fslengthok := fslength £ 2*((fnlines - 1)*3 • 3.5)

This physical constraint insures that the length of the flange splice plate is large enough for the bolt holes.

FSLOCOK fslocok := (fsloc is equal to slend in SEGMENTS)

This physical constraint insures that the flange splice is located at a section end.

FSPLICEOK fspliceok := afremok AND splicecapok AND book AND biok AND fslengthok AND fslocok

This constraint from the design code insures that all of the constraints for the flange splice part of the stiffener and splice design phase are satisfied.

GIRDEROK girderok := conceptok AND beamok AND stiffok AND fspliceok AND wspliceok

This design objective constraint insures that all constraints from all three design phases are satisfied.

GRADEOK gradeok := (grade is in GRADES)

This physical constraint insures that the grade of steel is valid (only the valid grades are contained in the GRADES relation.

HBSOK hbsok := ABS(h - hbs) £ TOLERANCE('hbsok')

This physical constraint insures that the bearing stiffener height is equal to the web height.

Figure 4-4: Constraints used in design example, continued.

HTOK — htok := h/tw £ 7500/((maxshear*1000)/(h*tw»$^{1/2}$

This constraint from the design code was chosen to satisfy the design objective that no transverse stiffeners be included in the design. It restricts the web slenderness ratio to be less than a specific maximum. The maxshear variable is the maximum shear stress in the girder. This constraint is exactly the same as CONHTOK except that it is used in the beam sizing phase after an analysis is performed and the exact value of maxshear is known.

IOK — iok := ABS(i - ix)/i £ TOLERANCE('iok')

This physical constraint insures that any changes to the web or flange dimensions between successive analyses are small so that the actual dimensions are close to the dimensions used for the analysis. The variable "i" is the calculated moment of inertia using the actual dimensions.

ISPLICEOK — ispliceok := isplice 2: ,75*tw*h^3/12

This constraint from the design code insures that the moment of inertia of the web splice plates (isplice) is not less than the required moment of inertia.

LENGTHOK — lengthok := absdength - sumlen) £ TOLERANCE('lengthok')

This physical constraint insures that the sum of the section lengths (sumlen) is equal to the girder length.

SOK — sok := sreqd £ .75*tws*bws^2*2/6

This constraint from the design code insures that the section modulus of the web splice plates is not less than the required section modulus (sreqd).

SPLICECAPOK — splicecapok := tencap ≤ f

This constraint from the design code insures that the actual force carried by the flange splice plates (f) is no greater than their capacity (tencap).

Figure 4-4: Constraints used in design example, continued.

STIFFOK stiffok := tbsok AND ccok AND forceok AND bearingok AND bbsok AND hbsok AND bslocok

This constraint from the design code insures that all of the constraints for the stiffener design part of the stiffener and splice design phase are satisfied.

STRESSOK stressok := fball £ m*c/i

This constraint from the design code insures that the actual stress in a particular section of the girder is no greater than the allowable stress. The variables "m", "c" and "i" are the bending moment, distance from the flange to the neutral axis and the moment of inertia of the section.

SUPPORTLOCOK supportlocok := (supportlocd) = 0) AND (supportloc(num) = length)

This physical constraint insures that the first support is located at the left end of the girder and the last support (num) is located at the right end of the girder.

TBSOK tbsok := tbs ≤ bbs*(fy/33000)$^{1/2}$/12

This constraint from the design code insures that the thickness of the bearing stiffener is no greater than a specific maximum limit (which is a function of the stiffener width).

WSLENGTHOK wslengthok := wslength £ 2*<(wnlines - 1)*3 + 3.5)

This physical constraint insures that the length of the web splice plate is large enough for the bolt holes.

WSLOCOK wslocok := (wsloc is equal to slend in SEGMENTS)

This physical constraint insures that the web splice is located at a section end.

WSPLICEOK wspliceok := sok AND awremok AND boltfok AND ispliceok AND bwsok AND wslengthok AND wslocok

This constraint from the design code insures that all of the constraints for the web splice part of the stiffener and splice design phase are satisfied.

Figure 4-4: Constraints used in design example, continued.

CONSTRAINT	PHASE(S)
AFREMOK	III
AWREMOK	III
BBSOK	III
BEAMOK	II
BEARINGOK	III
BIOK	III
BOLTFOK	III
BOOK	III
BSLOCOK	III
BWSOK	III
CCOK	III
CHANGEOK	I, II
CLEAROK	I, II, III
CONCEPTOK	I
CONFLANGEOK	I
CONHTOK	I
CONIFLANGEOK	I
CONIOK	I
CONNOK	I, II, III
DEFOK	II

Table 4-1: List of phase(s) that use each constraint.

CONSTRAINT	PHASE(S)
FNBOLTSOK	III
FORCEOK	IN
FSLENGTHOK	III
FSLOCOK	III
FSPLICEOK	III
GRADEOK	I, II, III
GIRDEROK	-
HBSOK	III
HTOK	II
IOK	II
ISPLICEOK	III
LENGTHOK	I, II, III
SOK	III
SPLICECAPOK	III
STIFFOK	III
STRESSOK	II
SUPPORTLOCOK	II, III
TBSOK	III
WSLENGTHOK	III
WSLOCOK	III
WSPLICEOK	III

Figure 4-1: List of phase(s) that use each constraint.

4.4. EXTERNAL PROCEDURES

Certain aspects of the design procedure are independent of the type of constraint enforcement method used. These aspects include the estimation of shears and moments for the conceptual design phase, the static analysis of the structure for the beam sizing phase and the selection of dimensions for all of the phases. In order to simplify the examples, these aspects are assumed to be performed by external procedures which can be called by the design program or the constraint evaluation procedures. It is assumed that these procedures have access to the database, and therefore, only a minimum of data need to be passed to them. A list of the necessary external procedures is shown in Figure 4-5. Table 4-2 provides a list of each procedure and the phase number in which it is used.

ANALYZE(trialid) Perform the static analysis of the specific design alternative (trialid) of the structure. This procedure retrieves the basic information about the structure, such as girder and section data, performs the analysis and loads the entire ANALYSIS relation.

CALCBOLTF1(boltf$_f$ vreqd, wnlines, wnbolts)
Calculate the force on each bolt (boltf) of the web splice for the normalized design example. The variable "vreqd" is the required shear stress.

CALCBOLTF2(boltf, wloc)
Calculate the force on each bolt (boltf) of the web splice located at wloc for the augmented design example. Notice the difference between this procedure and the one above. This procedure assumes that the data has already been stored in the database and that it can retrieve and calculate the necessary data to calculate boltf. The other procedure must be passed all of the necessary information.

CALCISPLICEKisplice, bws, tws, wnlines, wnbolts)
Calculate the moment of inertia of the web splice plate (isplice) for the normalized design example.

CALCISPLICE2(isplice, wloc)
Calculate the moment of inertia of the web splice plate (isplice) located at wloc for the augmented design example. Notice the difference between this procedure and the one above. This procedure assumes that the data has already been stored in the database and that it can retrieve and calculate the necessary data to calculate isplice. The other procedure must be passed all of the necessary information.

CALCMSHEAR(maxshear)
Compute the maximum shear in the structure (maxshear), using the analysis results.

CALCREACT(reaction, loc)
Calculate the reaction (reaction) at the support located at loc.

CALCSEGMENTS Recalculate the segment lengths and locations so that the connections are located at points of minimum moment.

Figure 4-5: External procedures used in design example.

DBMS(...) Perform any DBMS query language statement, which are specified in the argument of the procedure. This argument is considered a single transaction (it implies the BEGIN and END TRANSACTION commands in SEQUEL 2). Each operation returns a logical "actionok" variable.

ESTMOM(mom) Estimate the maximum moment in the girder to check coniok.

ESTNEGPOSMOM(negmom, posmom)
Estimate the maximum negative (negmom) and positive moments (posmom) to check coniflangeok.

ESTSHEAR(shear) Estimate the maximum shear force in the girder to check conhtok.

ON("<expression> (<action>)")
Perform the <action> whenever the expression is true. Used to handle abnormal DBMS transaction failures.

SIZEBFTF(bf, tf, quit)
Select the flange dimensions. The procedure will return a value of TRUE via the argument quit if it is not successful in generating a set of data. This quit option is provided in all of the following procedures.

SIZEBOTO(bo, to, bi, ti, quit)
Select the width and thickness of the outside and inside flange splice plates. The returned value of quit is True if the procedure did not successfully select new data.

SIZEBWSTWS(bws, tws, quit)
Select the width and thickness of the web splice plates. The returned value of quit is True if the procedure did not successfully select new data.

Figure 4-5: External procedures used in design example, continued.

SIZEFNUfnlines, fslength, fnbolts, quit)

Select the bolting data and the length for the flange splice plates. The returned value of quit is True if the procedure did not successfully select new data.

SIZEHTW<h, tw, quit)

Select the web dimensions. The returned value of quit is True if the procedure did not successfully select new data.

SIZETBSBBS(tbs, bbs, quit)

Select the bearing stiffener thickness and width. The returned value of quit is True, if the procedure did not successfully select new data.

SIZEWNUwslength, wnlines, wnbolts, quit)

Select the length and bolting data for the web splice plates. The returned value of quit is True if the procedure did not successfully select new data.

Figure 4-5: External procedures used in design example, continued.

PROCEDURE	PHASE(S)
ANALYZE	II
CALCBOLTF1	III
CALCBOLTF2	III
CALCISPLICE1	III
CALCISPLICE2	III
CALCMSHEAR	II
CALCREACT	III
CALCSEGMENTS	II
DBMS	I, II, III
ESTMOM	I
ESTNEGPOSMOM	I
ESTSHEAR	I
ON	I, II, III
SIZEBFTF	I, II
SIZEBOTO	III
SIZEBWSTWS	III
SIZEFNL	III
SIZEHTW	I. II
SIZETBSBBS	III
SIZEWNL	III

Table 4-2: List of phase(s) that use each procedure.

CHAPTER FIVE
CONSTRAINT PROCESSING USING DESIGN EXAMPLE

This chapter presents example computer subprogram-relational database implementations of the design procedure described in the previous chapter, using two of the constraint enforcement alternatives discussed in Section 3.4. Section 5.1 discusses the implementation using a completely normalized database that can only enforce single relation-single attribute constraints. Section 5.2 discusses the implementation using the proposed augmented database which has access to all of the retrieval operations so that all constraint types can be enforced.

5.1. NORMALIZED DESIGN DATABASE

This section presents segments of a computer program which implement the design procedure using a normalized relational database. Each segment performs one of the three design phases and is completely independent from the other segments. The database schema is the same as the one shown in Figure 4-2. Input and output data for each segment is contained in two specific sets of relations, which are shown for each phase.

5.1.1. CONCEPTUAL DESIGN PHASE

5.1.1.1. DATABASE SCHEMA

The conceptual design phase receives all input from the three relations shown in Figure 5-1.

Since most of the data contained in these three relations is selected by the designer (based on experience), they are completely loaded before this phase begins. At the beginning of the phase, the data in these relations are verified.

STRUCTURE(grade, fball, fv, e, detail, length, clear, boltdia, holedia, fbbolt)

SEGMENTSia/ternative, sect ion id, slength, slend, posmom)

GRADES(*grade*, fy)

Figure 5-1: Relations needed for input to phase I.

by checking the corresponding constraints. If any constraints are violated, no other operations are performed in this or subsequent phases.

All output from this phase is stored in the WSECTIONS and the FSECTIONS relations, shown in Figure 5-2.

WSECTIONS(a/ter/73f/Ve, h, tw)

FSECTI0NS(a/te/7?a?/Ve, *posmom,* bf, tf)

Figure 5-2: Relations needed for output from phase I.

5.1.1.2. ALGORITHM

The following steps are included in the conceptual design phase implementation.

1. Check the constraints on the summation of the section lengths and the girder length (lengthok), the connectivity of the sections (connok) and the validity of the grade of steel (gradeok).
2. Estimate shear and moment.
3. Repeat the following process until the constraints are satisfied and the values are successfully inserted into the database.
   a. Choose the web dimensions.
   b. Check the constraints on the web slendemess ratio (conhtok) and the moment of inertia of the web (coniok).
   c. Insert web dimensions if conhtok and coniok are satisfied.
4. Estimate the maximum moment for the positive and negative moment sections.
5. Repeat the following process until the constraints are satisfied and

the values are successfully inserted into the database for both moment sections.

a. Choose the flange dimensions.

b. Check clearance (clearok), flange moment of inertia (coniflangeok), flange dimensions (conflangeok) and change in section heights (changeok).

c. Insert flange dimensions if all constraints are satisfied.

The following Pascal-like procedure can be used to perform the conceptual design phase.

```
PROCEDURE CONCEPKtrialid, conceptok)
  BEGIN
    ONCNOT selectok (conceptok := 'false',
                          ERRORC'data missing for conceptual
                                design phase"),
                          RETURN)")

{ CHECK LENGTHOK }
      DBMS("SELECT(length from STRUCTURE,
                  SUM(slength) into sumlen from SEGMENTS
                  where alternative = trialid, selectok)")

      lengthok := absdength - sumlen) £ TOLERANCE('lengthok')

{ CHECK CONNOK }
      DBMS("SELECT(sectionid, slength, slend, COUNT into num from
                  SEGMENTS order by slend where alternative = trialid,
                  selectok)")

      IF (num # 0) AND (slendd) = 0) AND (slend(num) + slength(num) =
          length) THEN
        BEGIN
          connok := 'true'
          FOR j • 2 TO num
            BEGIN
              WHILE connok DO
                BEGIN
                  connok := abs(slend(j - 1) + slengtMj - 1) - slend(j))
                            £ TOLERANCE('connok')
                END
            END
        END
      ELSE
          connok :» 'false'

{ CHECK GRADEOK }
```

```
        DBMS("SELECT{grade into grade1 from GRADES where grade =
                    SELECT(grade from STRUCTURE), selectok)")

            gradeok := 'true'

        IF NOT {lengthok AND connok AND gradeok) THEN
          BEGIN
            ERRORC'lengthok or connok or gradeok failed")
            conceptok := 'false'
            RETURN
          END

        DBMS("SELECT(fball, from STRUCTURE,
                    fy from GRADES where grade = grade1, selectok)")


{ CHOOSE WEB DIMENSIONS }
        ESTSHEAR(shear)
        ESTMOM(mom)
        sreqd := mom*12/fball

        REPEAT     { CHOOSE WEB AND FLANGE DIMENSIONS UNTIL
                     CONSTRAINTS ARE SATISFIED }
            REPEAT        { CHOOSE WEB DIMENSIONS UNTIL CONSTRAINTS
                            ARE SATISFIED OR USER QUITS }
                SIZEHTWth, tw, quit)
                IF quit THEN
                  BEGIN
                    conceptok := FALSE
                    RETURN
                  END

                conhtok := h«««shear»1000)/(h»tw))^1/2/7500 <J tw

                coniok := ABS(h - (3»h»sreqd/(2*tw))^1/3) £ TOLERANCE('iok')

                IF conhtok AND coniok
                    DBMS("INSERT(into WSECTIONS(h, tw): <h, tw>
                                where alternative = trialid, insertok)")
            UNTIL conhtok AND coniok AND insertok

{ CHOOSE FLANGE DIMENSIONS }
            ESTNEGPOSMOMGiegmom, posmom)
            FOR j := 1 TO 2.   { CHOOSE FLANGE DIMENSIONS FOR }
              BEGIN            { POSITIVE AND NEGATIVE SEGMENTS }
                IF (j = 1) THEN
                    mom := negmom
                    pmom := FALSE
                ELSE
                    mom := posmom
                    pmom := TRUE
```

```
            ireqd := mom*h/(2*fball)
            iweb := tw*(h)^3/12
            iflanges := ireqd - iweb

            REPEAT      { CHOOSE FLANGE DIMENSIONS UNTIL
                          CONSTRAINTS ARE SATISFIED OR
                          USER QUITS }

                SIZEBFTF(bf, tf, quit)
                IF NOT quit THEN
                    DBMS("SELECT(clear from STRUCTURE, selectok)")
                    clearok := abs(h + tf*2 - clear) ≤
                               TOLERANCE('clearok')
                    coniflangeok := bf*tf*(d/2)^2*2 ≥ iflanges
                    conflangeok := bf/tf ≤ 65/(fy)^1/2
                    changeok := TRUE
                    IF j = 2 THEN
                      BEGIN     { CHECK ONLY IF BOTH
                                  SECTIONS ARE CHOSEN }
                        DBMS("SELECT(tf into tf2 from
                                     FSECTIONS where posmom = FALSE
                                     and alternative = trialid, selectok)")
                        changeok := abs(tf2 - tf) ≤
                                    TOLERANCE('changeok')
                      END

                    IF (coniflangeok AND conflangeok AND clearok AND
                        changeok) THEN
                        DBMS("INSERT(into FSECTIONS: <trialid, pmom,
                                     bf, tf>, insertok)")
                ELSE
                    conceptok := FALSE
                    RETURN

            UNTIL (coniflangeok AND conflangeok AND clearok AND
                   changeok AND insertok) OR quit
            IF NOT quit THEN
                momentok(j) := TRUE
            ELSE
                momentok(j) := FALSE
          END
        conceptok := momentok(1) AND momentok(2)
    UNTIL conceptok
END         { CONCEPT }
```

5.1.2. BEAM SIZING PHASE

5.1.2.1. DATABASE SCHEMA

The beam sizing phase receives all input from the STRUCTURE, SEGMENTS, WSECTIONS, FSECTIONS, GIRDER and SUPPORTS relations, shown In Figure 5-3.

STRUCTURE(grade, fball, fv, e, $defall_f$ length, clear, boltdia, holedia, fbbolt)

SEGMENTS(a/ternat7ve, sectionid, slength, slend, posmom)

WSECTI0NS(a/te/7*af/Ve, h, tw)

FSECTIONS(a/te/viaf/Ve, *posmom.* bf, tf)

G\RDER(a/ternative, numgirder)

SUPPORTS(a/te/77af/Ve, supportloc)

Figure 5-3: Relations needed for input to phase II.

Again, before beginning, all constraints corresponding to this input data are checked and the phase does not continue if these constraints are not satisfied. This phase calls an analysis procedure to perform the static analysis of the structure. The results of this procedure are stored in the ANALYSIS relation. The only other output from this phase consists of possible updates to the WSECTIONS, FSECTIONS or the ANALYSIS relation (if the analysis procedure is recalled). These three relations are shown in Figure 5-4.

ANALYSIS(a/te/v?af/Ve, *sectionid. load, analoc,* shear, mom, defy, rot, ix)

WSECT\ONS(a/ternative, h, tw)

FSECTI0NS(a/te/7?af/Ve, *posmom.* bf, tf)

Figure 5-4: Relations needed for output from phase II.

5.1.2.2. ALGORITHM

The following steps are included in the beam sizing design phase implementation.

1. Check the length (lengthok), connectivity (connok), clearance (clearok), change in height (changeok) and support location (supportlocok) constraints.

2. Perform the analysis, find the maximum shear force.

3. Check the constraint on the web slenderness (htok). If it is not satisfied, repeat the following process until the constraints are satisfied and the values are successfully inserted into the database.

    a. Select new web dimensions.

    b. Check the clearance (clearok) and and the web slenderness (htok) constraints.

    c. Update with the new values if clearok and htok are satisfied.

4. Repeat the following process for the positive and negative moment sections.

    a. Calculate the maximum moment

    b. Check the bending stress constraint (stressok). If the constraint fails, repeat the following process until the constraints are satisfied and the values are successfully inserted.

        i. Select new flange dimensions.

        ii. Check the bending stress (stressok), the clearance (clearok) and the change in section heights (changeok) constraints.

        iii. Update with the new values if these constraints are satisfied.

5. Check the deflection limit (defok) for each span. If the constraint fails, repeat the following process until the constraints are satisfied and the values are successfully inserted.

    a. Choose new web dimensions.

    b. Check the clearance constraint (clearok).

    c. Update with the new values if the clearok constraint is satisfied.

d. Choose new flange dimensions.

e. Check the clearance (clearok) and change in section heights (changeok) constraints.

f. Update with the new values if the clearok and the changeok constraints are satisfied.

6. Check the change in section dimension constraint (iok).

7. Check the beam satisfaction cōnstraint (beamok). If the constraint fails, and the user has not failed to select new dimensions, select new section locations based on zero moment points from the analysis and repeat the entire phase again.

The following procedure can be used to perform the beam sizing phase.

```
PROCEDURE BEAMSIZEUrialid, beamok)
  BEGIN
    ONC'NOT selectok (beamok := FALSE,
                      ERRORC'data missing for beam sizing phase"),
                      RETURN)")

    ONC'NOT joinok (beamok := FALSE,
                      ERRORC'error in ANALYSIS or SEGMENTS
                              relations - join failed"),
                      RETURN)")

    REPEAT      { CONTINUE UNTIL CONSTRAINTS ARE SATISFIED
                  OR USER QUITS }
{ CHECK LENGTHOK }
        DBMS("SELECT(length from STRUCTURE,
                  SUM(slength) into sumlen from SEGMENTS where
                  alternative = trialid, selectok)")

        lengthok := absdength - sumlen) £ TOLERANCE('lengthok')

{ CHECK CONNOK }
        DBMS("SELECT(sectionid, slength, slend, COUNT into num from
                  SEGMENTS order by slend where alternative =
                  trialid, selectok)")

        IF (num # 0) AND (slendd) = 0) AND (slend(num) +
            slength(num) = length) THEN
          BEGIN
            FOR j = 2 TO num
              BEGIN
                WHILE connok DO
                  BEGIN
                    connok := abs(slend(j - 1) + slength(j - 1) - slend(j))
                              ≤ TOLERANCE('connok')
```

```
                END
            END
        END
    ELSE
        connok := FALSE

{ CHECK CLEAROK }
    DBMS("SELECT(clear from STRUCTURE,
                h from WSECTIONS where alternative = trialid,
                tf from FSECTIONS where alternative = trialid and
                tf = MAX(tf). selectok)")

    clearok :• abs(h + tf»2 - clear) £ TOLERANCE('clearok')

{ CHECK CHANGEOK }
    DBMS("SELECT(tf into tf1 from FSECTIONS where posmom = TRUE
                and alternative • trialid,
                tf into tf2 from FSECTIONS where posmom = FALSE
                and alternative = trialid, selectok)")

    changeok := abs(tf2 - tf1) ^ TOLERANCE('changeok')

{ CHECK SUPPORTLOCOK }
    DBMS("SELECT(supportloc, COUNT into num from supports order
                by supportloc where alternative = trialid,
                selectok)")

    IF (num = 0) THEN
        supportlocok := FALSE
    ELSE
        supportlocok := (supportlocd) = 0) AND
                        (supportloc(num) = length)

    IF NOT (lengthok AND connok AND clearok AND changeok AND
            supportloc) THEN
      BEGIN
        ERRORC'lengthok or connok or clearok changeok or
                supportloc failed")
        beamok := FALSE
        RETURN
      END

    reanalyze := TRUE

{ PERFORM ANALYSIS }
    ANALYZE(trialid)

    CALCMSHEAR(maxshear)
    DBMS("SELECT(h, tw from WSECTIONS where alternative
                = trialid, selectok)")

{ CHECK WEB DIMENSIONS }
```

```
htok := h/t ≤ 7500/((maxshear*1000)/(h«tw))^1/2
IF NOT htok THEN
  BEGIN
    REPEAT    { CHOOSE NEW WEB DIMENSIONS UNTIL
                  CONSTRAINTS ARE SATISFIED OR
                  USER QUITS }
        SIZEHTWfh, tw, quit)
        IF NOT quit THEN
          BEGIN
            DBMS("SELECT(tf from FSECTIONS where alternative =
                  trial id and tf = MAX(tf), selectok)")

            clearok := abs(h + tf*2 - clear) £ TOLERANCECclearok')

            htok :• h/t £ 7500/((maxshear»1000)/(h«tw))^1/2
            IF htok AND clearok THEN
                DBMS("UPDATE<WSECTIONS(h, twh <h, tw> where
                      alternative = trialid, updateok)")
          END
    UNTIL (htok AND updateok AND clearok) OR quit
    IF quit THEN
        reanalyze := FALSE
  END


DBMS("SELECT(fball. defall from STRUCTURE, selectok)")

{ CHECK STRESS IN NEGATIVE AND POSITIVE MOMENT SECTION >
FOR j := 1 TO 2
  BEGIN    { CHECK CONSTRAINTS FOR POSITIVE
             AND NEGATIVE SECTIONS }
    IF (j = 1) THEN
        pmom := FALSE
    ELSE
        pmom := TRUE

    DBMS("SELECT{sectionid into SECTS from SEGMENTS where
                posmom = pmom and alternative = trialid,
                selectok)")
    DBMS("SELECT(mom, analoc into loc, sectionid into secid from
                ANALYSIS where load =
                'dload' and sectionid is in (SELECTisectionid
                from SECTS)) and mom =
                max(abs(mom)), selectok)")
    deadload := mom
    DBMS("SELECT(mom from ANALYSIS where
                load * 'dload' and sectionid = secid
                and analoc = loc and mom = max(abs(mom)),
                selectok)")
    liveload := mom
    m := deadload + liveload + .22*liveload
    DBMS("SELECT(h, tw from WSECTIONS where alternative
```

```
                    = trialid,
                    bf, tf from FSECTIONS where posmom =
                    pmom and alternative = trialid, selectok)")
            REPEAT     { CHECK STRESSOK UNTIL SATISFIED }

                c := (h + 2*tf)/2
                i := tw*(h/2)^3/12 + bf*(tf/2)^3/12 + bf*tf*(h/2 - tf/2)^2
                stressok(j) := fball ≥ m*c/i
                IF NOT stressok(j)
                  BEGIN
                    REPEAT  { CHOOSE NEW FLANGE DIMENSIONS UNTIL
                              CONSTRAINTS ARE SATISFIED OR
                              USER QUITS }
                        SIZEBFTF(bf, tf, quit)
                        clearok := abs(h + tf*2 - clear) ≤
                                   TOLERANCE('clearok')

                        DBMS("SELECT(tf into tf1 from FSECTIONS where
                                   posmom ≠ pmom and
                                   alternative = trialid, selectok)")
                        changeok := abs(tf - tf1) ≤ TOLERANCE('changeok')

                    UNTIL (clearok AND changeok) OR quit
                    IF quit THEN
                        reanalyze := FALSE
                    ELSE
                        DBMS("UPDATE(FSECTIONS(bf, tf): <bf, tf>
                                   where posmom = pmom and
                                   alternative = trialid,
                                   updateok)")
                  END
              UNTIL updateok OR quit

            END

          DBMS("SELECT(supportloc, COUNT into num from supports order
                    by supportloc where alternative = trialid,
                    selectok)
               JOIN(ANALYSIS(sectionid, defy) with SEGMENTS(sectionid,
                    slength, slend, posmom) into TEMP over sectionid
                    where alternative = trialid, joinok)")

{ CHECK THE DEFLECTION IN EACH SECTION FOR EACH SPAN }
          FOR j := 1 TO (num - 1)
            BEGIN
              l := supportloc(j + 1) - supportloc(j)
              DBMS("SELECT(sectionid into sid, defy, posmom into pmom
                    from TEMP where defy = MAX(defy) and
                    supportloc(j + 1) ≥ (slend + slength)
                    and supportloc(j) ≤ (slend), selectok)")
              defok := l/defall ≥ defy
              IF NOT defok THEN
```

```
                BEGIN
                  REPEAT    { CHOOSE NEW WEB DIMENSIONS }
                       SIZEHTW(h, tw, quit)
                       clearok := abs(h + tf*2 - clear) £ TOLERANCE('clearok')

                       DBMS("UPDATE<WSECTIONS<h, twh <h, tw> where
                                  alternative = trialid, updateok)")
                  UNTIL {updateok AND clearok) OR quit
                  IF quit THEN
                       reanalyze := FALSE

                  REPEAT    { CHOOSE NEW FLANGE DIMS. }
                       SIZEBFTF(bf, tf. quit)
                       clearok := abs(h + tf«2 - clear) £ TOLERANCE('clearok')
                       DBMS("SELECT(tf into tf1 from FSECTIONS where
                                  posmom i* pmom and    •
                                  alternative = trialid, selectok)")
                       changeok := absttf - tf1) £ TOLERANCE('changeok')

                       DBMS("UPDATE(FSECTIONS(bf, tf): <bf, tf> where
                                   posmom = pmom and alternative =
                                   trialid, updateok)")
                  UNTIL updateok OR quit
                  IF quit THEN
                       reanalyze := FALSE
                END
           END


{ CHECK FOR LARGE CHANGE IN i }
           DBMS<"SELECT(sectionid into sid, COUNT into num from
                        SEGMENTS where alternative = trialid,
                        h, tw from WSECTIONS where alternative •
                        trialid, selectok)")
           iok := TRUE
           FOR j = 1 TO num
             BEGIN
               WHILE (iok) DO
                 BEGIN
                   DBMS("SELECT(bf, tf from FSECTIONS where alternative =
                              trialid and posmom =
                              SELECT(posmom from SEGMENTS where
                              sectionid = sid(j)),
                              ix from ANALYSIS where
                              sectionid = sid(j), selectok)")
                   i := tw*(h/2)^3/12 + bf*(tf/2)^3/12 + bf*tf»(h/2 - tf/2)^2
                   iok := ABS(i - ix)/i £ TOLERANCE('iok')
                 END
             END

           beamok := htok AND stressokd) AND stressok(2) AND defok
                       and iok
```

```
        IF (NOT beamok) AND (reanalyze) THEN
            CALCSEGMENTS
    UNTIL beamok OR (NOT reanalyze)
END  { BEAMSIZE }
```

5.1.3. STIFFENER AND SPLICE DESIGN PHASE

5.1.3.1. DATABASE SCHEMA

The stiffener and splice design phase receives all input from the STRUCTURE, SEGMENTS, WSECTIONS, FSECTIONS, SUPPORTS, GRADES and ANALYSIS relation shown in Figure 5-5.

STRUCTURE(grade, fball, fv, e, defall, length, clear, boltdia, holedia, fbbolt)

SEGMENTS(a/te/773f/Ve, *sect'on/d,* slength, slend, posmom)

WSECTI0NS(a/te/7Wtf/Ve, h, tw)

FSECTI0NS(a/te/7?a*/Ve, *posmom,* bf, tf)

SUPPORTS(a/fe/v?af/Ve, supportloc)

ANALYSIS(a/te/v?af/Ve, *sectionid, load, analoc,* shear, mom, defy, rot, ix)

GRADES^acfe, fy)

Figure 5-5: Relations needed for input to phase III.

The supportloc attribute in the SUPPORTS relation is used to position the bearing stiffeners, since a stiffener is needed at each support. The slength and slend attributes of the SEGMENTS relation are used to find the splice locations.

The output from this phase is stored in the BSTIFFENERS, FSPLICES and WSPLICES relations shown in Figure 5-6.

BSTIFFENERS(*alternative, bsloc,* hbs, tbs, bbs)

FSPLICES(*alternative, fsloc,* bo, to, bi, ti, fslength, fnlines, fnbolts)

WSPLICES(*alternative, wsloc,* bws, tws, wslength, wnlines, wnbolts)

Figure 5-6: Relations needed for output from phase III.

5.1.3.2. ALGORITHM

The following steps are included in the stiffener and splice design phase implementation.

1. Check the constraints on the support locations (supportlocok), the clearance (clearok) and the grade of steel (gradeok).
2. For each support, repeat the following process until the constraints are satisfied and the values are successfully inserted into the database.
    a. Select stiffener dimensions.
    b. Check the constraints on the stiffener height (hbsok), width (bbsok), thickness (tbsok) and slenderness ratio (ccok).
    c. Calculate the reaction at the support.
    d. Check the constraints on the stiffener force (forceok), location (bslocok) and bearing capacity (bearingok).
    e. Insert the values.
3. Check the girder length (lengthok) and connectivity (connok) constraints.
4. Repeat the following process for each splice.
    a. Repeat the following process until the constraints are satisfied.
        i. Select bolting pattern and plate length for the flange splice.
        ii. Check the constraints on the number of bolts (fnboltsok), the length (fslength), the location of the flange splice (fslocok) and the remaining area (afremok) of the flange splice plate.
    b. Repeat the following process until the constraints are satisfied and the values are successfully inserted into the database.

i. Select flange splice plate widths and thicknesses.

ii. Check the constraints on the width of the outside flange splice plate (book), the width of the inside flange splice plate (biok) and the capacity of the splice plates (splicecapok).

iii. Insert all of the values if all flange splice constraints are satisfied.

c. Repeat the following process until the constraint is satisfied.

i. Select width and thickness of the web splice plates.

ii. Check the constraint on the section modulus of the splice plate (sok).

d. Repeat the following process until the constraints are satisfied and the values are successfully inserted into the database.

i. Select bolting pattern and plate length for the wed splice plates.

ii. Check the constraints on the width (bwsok), the length (wslength) and the remaining area (awremok) of the web splice.

iii. Calculate the force per bolt.

iv. Check the constraint on the force per bolt for the web splice (boltfok).

v. Calculate the moment of inertia of the splice plates.

vi. Check the constraints on the moment of inertia (ispliceok) and the location of the web splice plate (wslocok).

vii. Insert all of the new values if all web splice plate constraints are satisfied.

The following procedures can be used to perform the third phase of the design procedure.

```
PROCEDURE STIFF(trialid, stiffok)
  BEGIN
    ONC'NOT selectok (stiffok := FALSE,
                      ERRORC'data missing for stiffener design
                             phase"),
                      RETURN)")
```

```
{ CHECK SUPPORTLOCOK }
        DBMS("SELECT(length from STRUCTURE,
                    supportloc, COUNT into num from supports order by
                    supportloc where alternative = trialid, selectok)")

        supportlocok := (supportlocd) = 0) AND (supportloc(num) = length)

{ CHECK CLEAROK }
        DBMSC'SELECTOi, tw from WSECTIONS where alternative = trialid,
                    tf from FSECTIONS where alternative =
                    trialid and tf = MAX(tf), selectok)")

                    clearok := abs(h + tf»2 - clear) £ TOLERANCE('clearok')

{ CHECK GRADEOK }
        DBMS("SELECT(grade into grade1 from GRADES where grade =
                    SELECTtgrade from STRUCTURE), selectok))

                   gradeok := TRUE

        IF NOT (supportlocok AND clearok AND gradeok) THEN
          BEGIN
            ERRORC'SUPPORTLOCOK OR CLEAROK OR GRADEOK FAILED")
            stiffok := FALSE
            RETURN
          END

        DBMS("SELECT(e from STRUCTURE,
                    fy from GRADES where grade = grade1, selectok)")

{ CHOOSE STIFFENER DIMENSIONS }
        hbs := h
        FOR j := 1 to num DO
          BEGIN
            REPEAT    { CHOOSE THE STIFFENER DIMENSIONS UNTIL ALL
                        CONSTRAINTS ARE SATISFIED OR THE USER QUITS }
                SIZETBSBBSUbs, bbs, quit)
                IF quit THEN
                    stiffok := FALSE
                    RETURN
                ELSE
                  BEGIN
{ CHECK STIFFENER CONSTRAINTS }
                    DBMS("SELECT(bf from FSECTIONS where posmom =
                              SELECT(posmom from SEGMENTS where
                              ((slend + slength) £ supportloc(j + 1)
                              and  slend £ supportloc(j))),
                              selectok)")
                    hbsok := (ABS(h - hbs) £ 0)
                    bbsok := bbs £ (bf - tw)/2
                    tbsok(j) := tbs £ bbs#(fy/33000)^1/2/12
```

```
                    bsi := tbs«(2*bbs + tw)^3/12 + (tw*i8 - tbs)»tw^3/12
                    bsarea := (2*bbs + tw)*tbs + (tw»i8 - tbs)«tw
                    r := bsi/bsarea
                    cc := (2*»r^2»e/fy)^1/2
                    k := 1.0
                    ccok(j) := cc £ k*h/r
                    fa = 23580 - 1.03»(k«h/r)^2
                    p := fa*bsarea
                    CALCREACT(reaction, supportloc(j))
                    forceok(j) := reaction £ p
                    bearingok(j) := reaction/(2*tbs»bbs) £ .9«fy
                    DBMS("INSERT(into BSTIFFENERS(alternative, bsloc,
                                 hbs, tbs, bbsh <trialid, supportloc(j),
                                 hbs, tbs, bbs>, insertok)")

{ BSLOCOK IS AUTOMATICALLY SATISFIED SINCE BSLOC IS LOADED WITH
  SUPPORTLOC(j) }
                      bslocok := TRUE
                    END
              UNTIL tbsok(j) AND ccok(j) AND forceok(j) AND bearingok(j)
                    AND insertok AND bbsok AND hbsok AND bslocok
              stiffok := TRUE
            END     { STIFF }


    PROCEDURE SPLICESUrialid, fspliceok, wspliceok)
      BEGIN
        ONC'NOT selectok (fspliceok := FALSE,
                                  wspliceok := FALSE,
                                  ERRORC'data missing for splice design phase")
                                  RETURN)")

{ CHECK LENGTHOK }
        DBMSC'SELECTOength from STRUCTURE,
             SELECT(SUM(slength) into sumlen from SEGMENTS where
                    alternative = trialid, selectok)")

        lengthok := absilength - sumlen) ^ TOLERANCE('lengthok')

{ CHECK CONNOK }
        DBMS("SELECT{sectionid, slength, slend, COUNT into num from
                     SEGMENTS order by slend where alternative = trialid,
                     selectok)")

        IF (num * 0) AND (slendd) = 0) AND (slend(num) +
              slength(num) = length) THEN
          BEGIN
            FOR j » 2 TO num
              BEGIN
                WHILE connok DO
                  BEGIN
                    connok := abs(slend(j - 1) + slengtMj - 1) - slend(j))
```

```
                                £ TOLERANCE('connok')
                        END
                    END
                END
            ELSE
                connok := FALSE

            IF NOT (lengthok AND connok) THEN
              BEGIN
                ERRORC'lengthok or connok failed")
                fspliceok := FALSE
                wspliceok := FALSE
                RETURN
              END

            DBMS("SELECT(fball, fv, fbbolt, holedia from STRUCTURE,
                        h, tw from WSECTIONS where alternative = trialid,
                        sectionid into sid, slend, COUNT into num from
                        SEGMENTS order by slend where alternative = trialid,
                        selectok)")
            fspliceok :* TRUE
            wspliceok := TRUE
            FOR j := 2 TO num DO
              BEGIN
                DBMSC'SELECTltf into tf1 from FSECTIONS where alternative =
                            trialid and posmom =
                            SELECT(posmom from SEGMENTS where sectionid =
                            sid(j - 1)).
                            tf into tf2 from FSECTIONS where alternative =
                            trialid and posmom =
                            SELECT(posmom from SEGMENTS where sectionid =
                            sid(j)), selectok)")

{ CHOOSE SMALLER SECTION }
                IF tf1 £ tf2 THEN
                    section :» sid(j - 1)
                ELSE
                    section := sid(j)

                DBMS("SELECT(tf, bf from FSECTIONS where alternative = trialid
                            and posmom =
                            SELECT(posmom from SEGMENTS where sectionid
                            = section), selectok)")
                resize := TRUE
                REPEAT      { UNTIL ALL FLANGE SPLICE CONSTRAINTS
                              ARE SATISFIED }
                    af := tf*bf
                    tencap := af*.75»fball

                    REPEAT    { CHOOSE BOLT INFORMATION AND PLATE LENGTH
                                FOR FLANGE SPLICE UNTIL CONSTRAINTS ARE
                                SATISFIED OR USER QUITS }
```

```
                SIZEFNL(fnlines, fslength, fnbolts, quit)
                fnboltsok := ABS(fnbolts - tencap/fbbolt) ≤
                             TOLERANCE('fnboltsok')

                nbline := fnbolts/fnlines
                fslengthok := fslength ≥ 2*((fnlines - 1)*3 + 3.5)
                afremok := .75 ≤ (bf - nbline*holedia)/bf
            UNTIL (afremok AND fslengthok AND fnboltsok) OR quit
            IF quit THEN
                resize := 'false"
            ELSE
              BEGIN
                REPEAT  { CHOOSE PLATE WIDTHS AND THICKNESSES
                          UNTIL CONSTRAINTS ARE SATISFIED OR
                          USER QUITS }
                   SIZEBOTO(bo, to, bi, ti, quit)
                   book := (bo ≤ bf AND bo ≥ (tw +
                            (nbline + 2)*2))
                   biok := (bi ≤ (bf - tw)/2 AND bi ≥
                            (nbline/2 + 1)*2)
                   anet := bo*to + 2*bi*ti - nbline*holedia*(to + ti)
                   f := anet*fball
                   splicecapok := tencap ≤ f
                UNTIL (splicecapok AND book AND biok) OR quit
                IF quit THEN
                    resize := FALSE

                IF (afremok AND splicecapok) THEN
                    DBMS("INSERT(into FSPLICES: <trialid, slend(j),
                                 bo, to, bi, ti, fslength, fnlines,
                                 fnbolts>, insertok)")

{ FSLOCOK IS AUTOMATICALLY SATISFIED SINCE FSLOC IS LOADED WITH
  SLEND(j) }
                fslocok := TRUE
              END
          UNTIL (afremok AND splicecapok AND insertok AND fslocok) OR
                NOT resize

          IF NOT resize THEN
              fspliceok := FALSE

          vreqd := .75*tw*h*fv
          sreqd := .75* tw*h^2/6
          REPEAT   { UNTIL WEB SPLICE IS OK }

              REPEAT    { CHOOSE PLATE WIDTH AND THICKNESS UNTIL
                          CONSTRAINT ARE SATISFIED OR USER QUITS }

                 SIZEBWSTWS(bws, tws, quit)
                 sok := sreqd ≤ .75*tws*bws^2*2/6
```

```
                UNTIL sok OR quit
                IF quit THEN
                    resize := FALSE
                ELSE
                  BEGIN
                    SIZEWNL(wslength, wnlines, wnbolts, quit)
                    IF NOT quit
                      BEGIN
                        nbline := wnbolts/wnlines
                        bwsok := (bws £ h AND bws £
                                  (nbline - 1)*3.5 + 3)
                        wslengthok := wslength £ 2*{(wnlines - 1)*3 + 3.5)
                        awremok := .75 £ h*tw - nbline*holedia*tw/(h*tw)
                        CALCBOLTFi(boltf, vreqd, wnlines, wnbolts)
                        boltfok := boltf <= fbbolt
                        CALCISPLICEKisplice, bws, tws, wnlines, wnbolts)
                        ispliceok := isplice £ .75*tw«h^3/12
                        IF bwsok AND wslengthok AND awremok AND
                            boltfok THEN
                            DBMS("INSERT(into WSPLICES: <trialid, slend(j),
                                        bws, tws, wslength, wnlines,
                                        wnbolts>, insertok)")

{ WSLOCOK IS AUTOMATICALLY SATISFIED SINCE WSLOC IS LOADED WITH
   SLEND(j) }
                        wslocok := TRUE
                      END
                    ELSE
                      resize := FALSE
                  END
              UNTIL (sok AND awremok AND boltfok AND ispliceok AND
                     bwsok AND wslengthok AND insertok AND
                     wslocok) OR NOT resize
              IF NOT resize THEN
                  wspliceok := FALSE

          END

      END { SPLICES }
```

5.1.4. DISCUSSION OF THE NORMALIZED DESIGN DATABASE EXAMPLE

The four procedures presented in the previous section can be used to perform the three basic design phases using a completely normalized database. Each procedure passes a logical variable which *flags* the success or failure of the specific design process. Therefore, an overall bridge design program could call these procedures to perform the design of the girders and then check the value of girderok in the following manner:

```
    .
    .
    .
CONCEPT(trialid, conceptok)

IF conceptok THEN
  BEGIN
    BEAMSIZE(trialid, beamok)
    IF beamok THEN
      BEGIN
        STIFF(trialid, stiffok)
        SPLICES(trialid, fspliceok, wspliceok)
      END
  END
girderok := conceptok AND beamok AND stiffok AND fspliceok AND
            wspliceok

IF girderok THEN
    .
    .
{ CONTINUE THE BRIDGE DESIGN }
    .
    .
```

Since the database can only check single relation-single attribute constraints, the responsibility of insuring integrity within the database is placed on the design program. Therefore, every time a particular piece of data is retrieved from the database, all constraints that depend on this data must be checked to insure that the data item is valid.

Also, the design program is completely dependent upon the specific constraints that it must enforce. A slight change in a constraint or the introduction of a new constraint requires a major update of the design program.

The next section presents the design example using an augmented constraint checking database. In this example, the responsibility of integrity checking is placed upon the database system, which is shown to solve the problems listed above.

5.2. AUGMENTED DESIGN DATABASE

This section will present the computer procedures to add to the database system which enforce particular design constraints and the actual design procedures that implement the design procedure described in Section 4.1 with an augmented relational database.

5.2.1. DATABASE SCHEMA

The entire database schema for this example is shown in Figure 5-7. Notice that the relations are exactly the same as the ones shown in Figure 4-2 for the normalized database example, except that the constraint status attributes are added for each constraint. The choice of the exact location of each of these attributes is based on the location of the data needed to evaluate the constraint and the amount of redundancy caused by the addition of the attribute. For example, the status attribute for the constraint that limits the bending stress in a section (stressok) is stored in the FSECTIONS relation because the stressok constraint is used for each section of the girder (positive and negative). The only other relation that contains information about positive and negative sections is the SEGMENTS relation. However, this relation does not contain any of the information needed to check the constraint. Therefore, it is more logical to store the stressok attribute in the FSECTIONS relation. The supportlocok attribute could be stored in the SUPPORTS relation instead of in the GIRDER relation, since the data needed to check the constraint is stored in the SUPPORTS relation. However, if it is stored in the SUPPORTS relation, each supportlocok would have the same value for each support for a given alternative, since supportlocok only checks to see if a support is located at each end of the girder (nothing is said about each individual support). This would introduce unnecessary redundancy. Finally, the changeok attribute is stored in the FSECTIONS relation. Even though this introduces redundancy (since the value for changeok is the same for the positive and negative moment sections), it is assumed that the cost of this small redundancy is negligible compared to other issues, such as logical location and the fact that the only data needed to check the changeok constraint is located in the FSECTIONS relation. Therefore, placing the changeok attribute in the FSECTIONS relation makes the constraint a single relation constraint. In fact, since multiple relation constraints are more complex the the single relation

RELATIONS

STRUCTURE(grade, fball, fv, e, detail, length, clear, boltdia, holedia, fbbolt, gradeok)

SEGMENTS(a/te/7?af/Ve, *sect ion id,* slength, slend, posmom, connok)

WSECTI0NS(a/te/7?af/Ve, h, tw, conhtok, coniok, htok)

FSECTIONS(a/te/77af/Ve, *posmom.* bf, tf, clearok, coniflangeok, conflangeok, defok, iok, changeok, stressok)

ANALYSlSia/ternative, sectionid, load, analoc. shear, mom, defy, rot, ix)

G\RDER(a/ternative, numgirder, girderok, beamok, stiffok, wspliceok, fspliceok, conceptok, lengthok, supportlocok)

SUPPORTSia/ternative. supportloc)

BSTIFFENERS(a/te/77af/Ve, *bsloc,* hbs, tbs, bbs, bbsok, bearingok, bslocok, hbsok, bbsok, tbsok, ccok, forceok)

FSPUCESia/ternative. fs/oc, bo, to, bi, ti, fslength, fnlines, fnbolts, afremok, biok, book, fslocok, fslengthok, splicecapok)

WSPL\ICES(a/ternative, ws/oc, bws, tws, wslength, wnlines, wnbolts, awremok, boltfok, bwsok, wslocok, sok, wslengthok, wspliceok)

GRADES(grade. fy)

SINGLE RELATION-SINGLE ATTRIBUTE CONSTRAINTS

0 < fball £ MAXfball	0 < clear £ MAXclear
0 < fv £ MAXfv	0 < boltdia £ MAXboltdia
0 < e £ MAXe	0 < holedia £ MAXholedia
0 < defall £ MAXdefall	0 < fbbolt £ MAXfbbolt

Figure 5-7: Database schema for augmented design database example.

0 < length £ MAXlength

0 < slend £ MAXslend

0 < tw £ MAXtw

0 < tf £ MAXtf

0 < numgirder £ MAXnumgirder

0 < bsloc £ MAXbsloc

0 < tbs £ MAXtbs

0 < fsloc £ MAXfsloc

0 < to £ MAXfo

0 < ti £ MAXti

0 < fnlines £ MAXfnlines

0 < wsloc £ MAXwsloc

0 < tws £ MAXtws

0 < wnlines £ MAXwnlines

0 < fy ≤, MAXfy

0 < slength £ MAXslength

0 < h £ MAXh

0 < bf £ MAXbf

0 < analoc £ 1.0

0 < supportloc £ MAXsupportloc

0 < hbs £ MAXhbs

0 < bbs £ MAXbbs

0 < bo £ MAXbo

0 < bi ≤ MAXbi

0 < fslength £ MAXfslength

0 < fnbolts £ MAXfnbolts

0 < bws £ MAXbws

0 < wslength £ MAXwsiength

0 < wnbolts £ MAXwnbolts

Figure 5-7: Database schema for augmented design database example, continued.

constraints, it is important to try to place the attribute such that the constraint becomes a single relation constraint. This same evaluation process was used to position all of the other constraint status attributes.

5.2.2. CONSTRAINT PROCEDURES

The following constraints are to be included in the augmented relational database.

```
    FUNCTION LENGTHOK(trialid)
{ CHECK LENGTHOK FOR THE GIRDER }
      BEGIN
        ONC'NOT selectoMlengthok := 'false',
                                ERRORC'data missing for lengthok"),
                                RETURN)")

        DBMSC'SELECTOengthfrom STRUCTURE,
                    SUM(slength) into sumlen from SEGMENTS where
                    alternative = trialid, selectok)")

        lengthok := absdength - sumlen) £ TOLERANCEC lengthok')
      END


    FUNCTION CONNOK(trialid, section)
{ CHECK CONNOK FOR SECTIONID = SECTION }
      BEGIN
        ONC'NOT selectoMconnok := 'false',
                                ERRORC'data missing for connok"),
                                RETURN)")

        DBMS("SELECT(slength into seclength, slend into lend from SEGMENTS
                    where alternative = trialid and sectionid = section,
                    selectok)")

        DBMSC'SELECTOengthfrom STRUCTURE,
                    slength, slend, COUNT into num from SEGMENTS order
                    by slend where alternative = trialid and
                    (slend = (lend + seclength) or (slend + slength) =
                    lend), selectok)")

        IF (num = 0 AND lend = 0 AND seclength = length) THEN
            connok := 'true'
        ELSE
            IF (num = 1) AND ((lend = 0 AND (slength + slend) = length) or
                  (slend = 0 AND (seclength + lend) = length)) THEN
                connok := 'true'
            ELSE
              IF (num = 2) THEN
```

```
                connok := (abs(slendd) + slengthd) - lend) £
                          TOLERANCECconnok')
                      AND ABSOend + seclength - slend(2» ≤,
                          TOLERANCECconnok'))
            ELSE
                connok := 'false'
        END


    FUNCTION GRADEOK
{ CHECK GRADEOK FOR THE STRUCTURE }
        BEGIN
          ONC'NOT selectok (gradeok := 'false'.
                          ERRORC'data missing for gradeok"),
                          RETURN)")

          DBMS("SELECT(grade from GRADES where grade =
                    SELECT(grade from STRUCTURE), selectok)")


          gradeok := 'true'
        END


    FUNCTION CONHTOK(trialid)
{ CHECK CONHTOK FOR THE GIRDER }
        BEGIN
          ONC'NOT selectok (conhtok := 'false'.
                          ERRORC'data missing for conhtok"),
                          RETURN)")

          DBMS(SELECTC'fball, from STRUCTURE.
                    h, tw from WSECTIONS where alternative = trialid,
                    selectok)")

          ESTSHEAR(shear)
          conhtok := h»((shear»1000)/(h*tw))^1/2/7500 £ tw
        END


    FUNCTION CONIOK(trialid)
{ CHECK CONIOK FOR THE GIRDER }
        BEGIN
          ONC'NOT selectok (coniok := 'false',
                          ERRORC'data missing for coniok"),
                          RETURN)")

          DBMS("SELECT(h, tw from WSECTIONS where alternative = trialid,
                    fball from STRUCTURE, selectok)")

          ESTMOM(mom)
          sreqd := mom*12/fball
```

```
      coniok := ABS(h - (3*h*sreqd/(2*tw))^1/3) ≤ TOLERANCE('coniok')
    END


  FUNCTION CONIFLANGEOK(trialid, pmom)
{ CHECK CONIFLANGEOK FOR THE SECTION WHERE POSMOM = PMOM }
    BEGIN
      ON("NOT selectok (coniflangeok := 'false',
                        ERROR("data missing for coniflangeok"),
                        RETURN)")

      DBMS("SELECT(bf, tf from FSECTIONS where alternative = trialid and
                   posmom = pmom,
                   h, tw from WSECTIONS where alternative = trialid,
                   fball from STRUCTURE, selectok)")

      ESTMOM(mom)
      ireqd := mom*h/(2*fball)
      iweb := tw*(h)^3/12
      iflanges := ireqd - iweb
      coniflangeok := bf*tf*(d/2)^2*2 ≥ iflanges
    END


  FUNCTION CONFLANGEOK(trialid, pmom)
{ CHECK CONFLANGEOK FOR THE SECTION WHERE POSMOM = PMOM }
    BEGIN
      ON("NOT selectok (conflangeok := 'false',
                        ERROR("data missing for conflangeok"),
                        RETURN)")

      DBMS("SELECT(bf, tf from FSECTIONS where alternative = trialid and
                   posmom = pmom,
                   fy from GRADES where grade = SELECT(grade from
                   STRUCTURE), selectok)")

      flangeok := bf/tf ≤ 65/(fy)^1/2
    END


  FUNCTION CLEAROK(trialid, pmom)
{ CHECK CLEAROK FOR THE SECTION WHERE POSMOM = PMOM }
    BEGIN
      ON("NOT selectok (clearok := 'false',
                        ERROR("data missing for clearok"),
                        RETURN)")

      DBMS("SELECT(clear from STRUCTURE,
                   h from WSECTIONS where alternative = trialid,
                   tf from FSECTIONS where alternative = trialid and
                   posmom = pmom, selectok)")
```

```
      clearok := abs(h + tf*2 - clear) £ TOLERANCE('clearok')
    END


  FUNCTION CHANGEOK(trialid)
{ CHECK CHANGEOK FOR THE GIRDER }
    BEGIN
      ONC'NOT selectok (changeok := 'false',
                        ERRORC'data missing for changeok"),
                        RETURN)")

      DBMS("SELECT(tf into tf1 from FSECTIONS where posmom = 'true'
                  and alternative = trialid,
                  tf into tf2 from FSECTIONS where posmom = 'false'
                  and alternative = trialid, selectok)")

      changeok := abs(tf2 - tf1) £ TOLERANCE('changeok')
    END


  FUNCTION SUPPORTLOCOK(trialid)
{ CHECK SUPPORTLOCOK FOR THE GIRDER }
    BEGIN
      ONC'NOT selectok (supportlocok := 'false',
                        ERRORC'data missing for supportlocok"),
                        RETURN)")

      DBMS("SELECT{supportloc. COUNT into num from supports order by
                  supportloc where alternative = trialid, selectok)")

      IF (num = 0) OR (num = 1) THEN
          supportlocok := 'false'
      ELSE
          supportlocok := (supportlocd) = 0) AND (supportloc(num) = length)
    END


  FUNCTION HTOK(trialid)
{ CHECK HTOK FOR THE GIRDER }
    BEGIN
      ONC'NOT selectok (htok := 'false',
                        ERRORC'data missing for htok"),
                        RETURN)")

          CALCMSHEAR(maxshear)
          DBMS("SELECT(h, tw from WSECTIONS where alternative
                      = trialid. selectok)")

          htok := h/tw ≤ 7500/((maxshear*1000)/(h*tw))^1/2
    END
```

```
    FUNCTION STRESSOK(trialid, section)
{ CHECK STRESSOK FOR THE SECTION WHERE SECTIONID = SECTION }
      BEGIN
        ONC'NOT selectok (stressok := 'false',
                          ERRORC'data missing for stressok"),
                          RETURN)")

            DBMS("SELECT(fball. defall from STRUCTURE,
                  posmom into pmom from SEGMENTS where
                  alternative = trialid and sectionid = section,
                  mom, analoc into loc, sectionid into secid from
                  ANALYSIS where load = 'dload' and sectionid
                  = section and mom =  max(abs(mom)), selectok)")

             deadload := mom

             DBMS(SELECT(mom from ANALYSIS where load # 'dload' and
                  sectionid = section and analoc s loc and mom
                  = max(abs(mom)),
                  h, tw from WSECTIONS where alternative = trialid,
                  bf, tf from FSECTIONS where posmom = pmom and
                  alternative = trialid, selectok)")

                liveload := mom
                m := deadload + liveload + .22«liveload -
                c := (h + 2*tf)/2
                i := tw»(h/2)^3/12 • bf»(tf/2)^3/12 + bf#tf*(h/2 - tf/2)^2
                stressok := fball £ m*c/i
      END


    FUNCTION DEFOK(trialid, section)
{ CHECK DEFOK FOR THE SECTION WHERE SECTIONID = SECTION }
      BEGIN
        ONC'NOT selectok (defok := 'false',
                          ERRORC'data missing for defok"),
                          RETURN)")

             DBMS("SELECT(defall from STRUCTURE,
                   defy from ANALYSIS where defy = MAX(defy) and
                   sectionid = section,
                   slend, slength from SEGMENTS where alternative =
                   trialid and sectionid = section, selectok)")

             DBMS("SELECT(MIN(supportloc) into minloc from supports
                   where supportloc £ (slend + slength),
                   MAX(supportloc) into maxloc from supports
                   where supportloc £ (slend), seJectok)")

             I := minloc - maxloc

             defok := I/defall £ defy
```

```
    END


  FUNCTION IOK(trialid, section)
{ CHECK IOK FOR THE SECTION WHERE SECTIONID = SECTION }
    BEGIN
      ONC'NOT selectok (iok := 'false',
                              ERRORC'data missing for iok"),
                              RETURN)")

      DBMS("SELECT(h, tw from WSECTIONS where alternative = trialid,
                  bf, tf from FSECTIONS where alternative » trialid and
                  posmom = SELECT(posmom from SEGMENTS where
                  sectionid = section),
                  ix from ANALYSIS where sectionid = section, selectok)")

      i :• tw»(h/2)^3/12 + bf*(tf/2)^3/12 • bf«tf»<h/2 - tf/2)^2
      iok := ABS(i - ix)/i £ TOLERANCE('iok')
  END


  FUNCTION BSLOCOKUrialid, bloc)
{ CHECK BSLOCOK FOR THE BEARING STIFFENER WHERE BSLOC = BLOC }
    BEGIN
      ONC'NOT selectok (bslocok := 'false',
                              ERRORC'data missing for bslocok"),
                              RETURN)")

      DBMS("SELECT(supportloc from SUPPORTS where alternative = trialid
                  and supportloc = bloc, selectok)")

          bslocok := 'true'
    END


  FUNCTION HBSOK(trialid, bloc)
{ CHECK HBSOK FOR THE STIFFENER WHERE BSLOC = BLOC }
    BEGIN
      ONC'NOT selectok (hbsok := 'false',
                              ERRORC'data missing for hbsok"),
                              RETURN)")

      DBMS("SELECT(h from WSECTIONS where alternative = trialid,
                  hbs from BSTIFFENERS where alternative = trialid and
                  bsloc = bloc, selectok)")

      hbsok := (ABS(h - hbs) £ TOLERANCE('hbsok')
    END


  FUNCTION BBSOKUrialid, bloc)
{ CHECK BBSOK FOR THE STIFFENER WHERE BSLOC = BLOC }
```

```
      BEGIN
        ONC'NOT selectok (bbsok := 'false',
                                ERRORC'data missing for bbsok"),
                                RETURN)")

        DBMS("SELECT(tw from WSECTIONS where alternative = trialid,
                     bf from FSECTIONS where posmom = SELECT(posmom
                     from SEGMENTS where (slend + slength) £ bloc
                     and (slend) £ bloc),
                     bbs from BSTIFFENERS where alternative = trialid and
                     bsloc = bloc, selectok)")

        bbsok := bbs £ (bf - tw)/2
      END


    FUNCTION TBSOK(trialid, bloc)
{ CHECK TBSOK FOR THE STIFFENER WHERE BSLOC = BLOC }
      BEGIN
        ONC'NOT selectok (tbsok := 'false',
                                ERRORC'data missing for tbsok"),
                                RETURN)")

        DBMS("SELECT(fy from GRADES where grade = SELECKgrade from
                     STRUCTURE),
                     tbs, bbs from BSTIFFENERS where alternative = trialid
                     and bloc = bsloc, selectok)")

        tbsok :» tbs £ bbs*(fy/33000)^1/2/12
      END


    FUNCTION CCOK(trialid, bloc)
{ CHECK CCOK FOR THE STIFFENER WHERE BSLOC * BLOC }
      BEGIN
        ONC'NOT selectok (ccok := 'false',
                                ERRORC'data missing for ccok"),
                                RETURN)")

        DBMS("SELECT(tw from WSECTIONS where alternative = trialid,
                     tbs, bbs, hbs from BSTIFFENERS where alternative =
                     triaiid and bsloc = bloc,
                     fy from GRADES where grade » SELECT(grade from
                     STRUCTURE),
                     e from STRUCTURE, selectok)")

            bsi := tbs«(2»bbs + tw)^3/12 + (tw*18 - tbs)*tw^3/12
            bsarea := (2*bbs + tw)*tbs + (tw*i8 - tbs)«tw
            r := bsi/bsarea
            cc := (2»ir^2*e/fy)^1/2
            k := 1.0
            ccok :* cc ^ k*hbs/r
```

```
    END


  FUNCTION FORCEOK(trialid, bloc)
{ CHECK FORCEOK FOR THE STIFFENER WHERE BSLOC = BLOC }
    BEGIN
      ONC'NOT selectok {forceok := 'false',
                       ERRORC'data missing for forceok"),
                       RETURN)")

      DBMSC'SELECTUw from WSECTIONS where alternative = trialid,
            tbs, bbs, hbs from BSTIFFENERS where alternative =
            trialid and bsloc = bloc, selectok)")

      bsi := tbs*(2*bbs + tw)^3/12 + (tw*18 - tbs)»tw^3/12
      bsarea :« (2*bbs + tw)*tbs + (tw*18 - tbs)*tw
      r := bsi/bsarea
      k := 1.0
      fa = 23580 - 1.03«(k»hbs/r)^2
      p := fa«bsarea
      CALCREACT<reaction, bloc)
      forceok := reaction ^ p
    END


  FUNCTION BEARINGOK(trialid. bloc)
{ CHECK BEARINGOK FOR THE STIFFENER WHERE BSLOC = BLOC }
    BEGIN
      ONC'NOT selectok (bearingok := 'false',
                       ERRORC'data missing for bearingok"),
                       RETURN)")

       DBMS(SELECT(tbs, bbs from BSTIFFENERS where alternative =
            trialid and bsloc = bloc,
            fy from GRADES where grade = SELECT(grade from
            STRUCTURE), selectok)")

       CALCREACT(reaction, bloc)
       bearingok := reaction/(2«tbs*bbs) £ .9«fy
    END


  FUNCTION FSLOCOKUrialid, floe)
{ CHECK FSLOCOK FOR THE FLANGE SPLICE WHERE FSLOC = FLOC }
    BEGIN
      ONC'NOT selectok (fslocok := 'false',
                       ERRORC'data missing for fslocok"),
                       RETURN)")

      DBMS("SELECT(slend from SEGMENTS where alternative = trialid and
            slend = floe, selectok)")
```

```
            fslocok := 'true'
        END


    FUNCTION FSLENGTHOKUrialid, floe)
{ CHECK FSLENGTH FOR THE FLANGE SPLICE WHERE FSLOC = FLOC }
        BEGIN
          ONC'NOT selectok (fslengthok := 'false',
                                ERRORC'data missing for fslengthok"),
                                RETURN)")

          OBMS("SELECT(fslength, fnlines from FSPLICES where alternative =
                        trialid and fsloc = floe, selectok)")

                       fslengthok := fslength £ 2»<(fnlines - 1)»3 + 3.5)
        END


    FUNCTION FNBOLTSOK<trialid, floe)
{ CHECK FNBOLTSOK FOR THE FLANGE SPLICE WHERE FSLOC = FLOC }
        BEGIN
          ONC'NOT selectok (fnboltsok := 'false',
                                ERRORC'data missing for fnboltsok"),
                                RETURN)")

          DBMSC'SELECT(fball, fbbolt from STRUCTURE,
                        tf, bf from FSECTIONS where alternative = trialid and
                        (slend + slength) £ floe and slend £ floe
                        and tf = MIN(tf),
                        fnbolts from FSPLICES where
                        alternative = trialid and fsloc = floe, selectok)")

          af := tf*bf
          tencap := .75*af»fball
          fnboltsok := ABS(fnbolts - tencap/fbbolt) £ TOLERANCE('fnboltsok')
        END


    FUNCTION AFREMOK(trialid, floe)
{ CHECK AFREMOK FOR THE FLANGE SPLICE WHERE FSLOC = FLOC }
        BEGIN
          ONC'NOT selectok (afremok := 'false',
                                ERRORC'data missing for afremok"),
                                RETURN)")

          DBMSC'SELECT(holedia from STRUCTURE,
                        bf from FSECTIONS where alternative = trialid and
                        (slend + slength) £ floe and slend £ floe
                        and tf = MIN(tf),
                        fnbolts, fnlines from FSPLICES where alternative = trialid
                        and fsloc = floe, selectok)")
```

```
      nbline := fnbolts/fnlines
      afremok := 75 ≤ (bf - nbline*holedia)/bf
    END


  FUNCTION BOOK(trialid, floc)
{ CHECK BOOK FOR THE FLANGE SPLICE WHERE FSLOC = FLOC }
    BEGIN
      ON("NOT selectok (book := 'false',
                        ERROR("data missing for book"),
                        RETURN)")

      DBMS("SELECT(bf from FSECTIONS where alternative = trialid and
                   (slend + slength) ≥ floc and slend ≤ floc,
                   bo, fnbolts, fnlines from FSPLICES where alternative =
                   trialid and fsloc = floc,
                   tw from WSECTIONS where alternative = trialid,
                   selectok)")

      nbline := fnbolts/fnlines
      book := (bo ≤ bf AND bo ≥ (tw + (nbline + 2)*2))
    END


  FUNCTION BIOK(trialid, floc)
{ CHECK BIOK FOR THE FLANGE SPLICE WHERE FSLOC = FLOC }
    BEGIN
      ON("NOT selectok (biok := 'false',
                        ERROR("data missing for biok"),
                        RETURN)")

      DBMS("SELECT(bf from FSECTIONS where alternative = trialid and
                   (slend + slength) ≥ floc and slend ≤ floc,
                   bi, fnbolts, fnlines from FSPLICES where alternative =
                   trialid and fsloc = floc,
                   tw from WSECTIONS where alternative = trialid,
                   selectok)")

      nbline := fnbolts/fnlines
      biok := (bi ≤ (bf - tw)/2 AND bi ≥ (nbline/2 + 1)*2)
    END


  FUNCTION SPLICECAPOK(trialid, floc)
{ CHECK SPLICECAPOK FOR THE FLANGE SPLICE WHERE FSLOC = FLOC }
    BEGIN
      ON("NOT selectok (splicecapok := 'false',
                        ERROR("data missing for splicecapok"),
                        RETURN)")

      DBMS("SELECT(holedia, fball from STRUCTURE,
                   tf, bf from FSECTIONS where alternative = trialid and
```

```
                    (slend + slength) ≥ floc and slend ≤ floc
                    and tf = MIN(tf),
                    bo, to, ti , bi, fnbolts, fnlines from FSPLICES where
                    alternative = trialid and fsloc = floc, selectok)")

      nbline := fnbolts/fnlines

      af := tf*bf
      tencap := .75*af*fball
      anet := bo*to + 2*bi*ti - nbline*holedia*(to + ti)
      f := anet*fball
      splicecapok := tencap ≤ f
    END


  FUNCTION WSLOCOK(trialid, wloc)
{ CHECK WSLOCOK FOR THE WED SPLICE WHERE WSLOC = WLOC }
    BEGIN
      ON("NOT selectok (wslocok := 'false',
                        ERROR("data missing for wslocok"),
                        RETURN)")

      DBMS("SELECT(slend from SEGMENTS where alternative = trialid and
                    slend = wloc, selectok)")

        wslocok := 'true'
    END


  FUNCTION SOK(trialid, wloc)
{ CHECK SOK FOR THE WEB SPLICE WHERE WSLOC = WLOC }
    BEGIN
      ON("NOT selectok (sok := 'false',
                        ERROR("data missing for sok"),
                        RETURN)")

      DBMS("SELECT(h, tw from WSECTIONS where alternative = trialid,
                    tws, bws from WSPLICES where alternative = trialid and
                    wsloc = wloc, selectok)")

      sreqd := .75* tw*h^2/6
      sok := sreqd ≤ .75*tws*bws^2*2/6
    END


  FUNCTION BWSOK(trialid, wloc)
{ CHECK BWSOK FOR THE WEB SPLICE WHERE WSLOC = WLOC }
    BEGIN
      ON("NOT selectok (bwsok := 'false',
                        ERROR("data missing for bwsok"),
                        RETURN)")
```

```
        DBMSC'SELECKbws, wnbolts, wnlines from WSPLICES where
                    alternative = trialid and wsloc = wloc,
                    h from SEGMENTS where alternative = trialid, selectok)")

        nbline := wnbolts/wnlines
        bwsok := (bws £ h AND bws £ (nbline - 1)*3.5 + 3)
      END


    FUNCTION WSLENGTHOK(trialid, wioc)
{ CHECK WSLENGTHOK FOR THE WEB SPLICE WHERE WSLOC = WLOC }
      BEGIN
        ONC'NOT selectok {wslengthok := 'false',
                                ERRORC'data missing for wslengthok"),
                                RETURN)")

        DBMS("SELECT(wslength, wnlines from WSPLICES where alternative =
                    trialid and wsloc = wloc, selectok)")

        wslengthok := wslength £ 2*<(wnlines - 1)*3 + 3.5)
      END


    FUNCTION AWREMOKUriaiid, wloc)
{ CHECK AWREMOK FOR THE WEB SPLICE WHERE WSLOC = WLOC }
      BEGIN
        ONC'NOT selectok (awremok := 'false',
                                ERRORC'data missing for awremok"),
                                RETURN)")

        DBMSC'SELECT{holedia from STRUCTURE,
                    wnbolts, wnlines from WSPLICES where alternative =
                    trialid and wsloc = wloc,
                    h, tw from WSECTIONS where alternative = trialid,
                    selectok)")

        nbline := wnbolts/wnlines
        awremok := .75 £ h«tw - nbline«holedia*tw/(h*tw)
      END


    FUNCTION BOLTFOK(trialid, wloc)
{ CHECK BOLTFOK FOR THE WEB SPLICE WHERE WSLOC = WLOC }
      BEGIN
        ONC'NOT selectok (boltfok :• 'false',
                                ERRORC'data missing for boltfok"),
                                RETURN)")

        DBMS("SELECT(fbbolt from STRUCTURE, selectok)")

        CALCBOLTF2(boltf, wloc)
        boltfok := boltf £ fbbolt
```

```
    END


  FUNCTION ISPLICEOKUrialid, wloc)
{ CHECK ISPLICEOK FOR THE WEB SPLICE WHERE WSLOC = WLOC }
    BEGIN
      ONC'NOT selectok (ispliceok := 'false',
                              ERRORC'data missing for ispliceok"),
                              RETURN)")

      DBMS("SELECT(h. tw from WSECTIONS where alternative = trialid,
            selectok)")

      CALCISPLICE2(isplice, wloc)
      ispliceok := isplice £ .75«tw»h^3/12
    END


  FUNCTION CONCEPTOK(trialid)
{ CHECK CONCEPTOK FOR THE STRUCTURE }
    BEGIN
      ONC'NOT selectok (conceptok := 'false',
                              ERRORC'data missing for conceptok"),
                              RETURN)")

      DBMS("SELECT(COUNT into num from WSECTIONS where (conhtok
                  = 'false' or coniok = 'false') and alternative = trialid,
                  COUNT into num1 from FSECTIONS where (coniflangeok
                  = 'false' or conflangeok = 'false') and alternative
                  = trialid, selectok)")

      conceptok := (num « 0) AND (num1 = 0)
    END


  FUNCTION BEAMOK(trialid)
{ CHECK BEAMOK FOR THE STRUCTURE }
    BEGIN
      ONC'NOT selectok (beamok := 'false',
                              ERRORC'data missing for beamok"),
                              RETURN)")

      DBMSC'SELECT(COUNT into num from WSECTIONS where htok = 'false'
                  and alternative = trialid,
                  COUNT into num1 from FSECTIONS where (stressok =
                  'false' or defok - 'false' or iok = 'false') and
                  alternative = trialid), selectok")

      beamok :« (num = 0) AND (num1 = 0)
    END
```

```
    FUNCTION STIFFOK(trialid)
{ CHECK STIFFOK FOR THE STRUCTURE }
      BEGIN
        ONC'NOT selectok (stiffok := 'false',
                                ERRORC'data missing for stiffok"),
                                RETURN)")

        DBMS("SELECT{COUNT into num from BSTIFFENERS where (tbsok =
              'false' or ccok ='false' or forceok ='false' or bearingok
              = 'false' or bbsok ='false' or hbsok ='false' or bslocok
              = 'false') and alternative = trialid, selectok)")

        stiffok :* (num = 0)
      END


    FUNCTION FSPLICEOK(trialid)
{ CHECK FSPLICEOK FOR THE STRUCTURE }
      BEGIN
        ONC'NOT selectok {fspiiceok := 'false',
                                ERRORC'data missing for fspiiceok"),
                                RETURN)")

        DBMS("SELECT(COUNT into num from FSPLICES where (afremok ='false'
              or splicecapok ='false' or book ='false' or biok ='false' or
              fslengthok ='false' or fslocok ='false') and alternative *
              trialid, selectok)")

        fspiiceok := (num = 0)
      END


    FUNCTION WSPLICEOK(trialid)
{ CHECK WSPLICEOK FOR THE STRUCTURE }
      BEGIN
        ONC'NOT selectok (wspliceok := 'false',
                                ERRORC'data missing for wspliceok"),
                                RETURN)")

        DBMSC'SELECT(COUNT into num from WSPLICES where (sok ='false' or
              awremok ='false' or boltfok ='false' or ispliceok ='false' or
              bwsok s'false' or wslengthok ='false' or wslocok ='false')
              and alternative = trialid, selectok)")

        wspliceok := (num = 0)
      END


    FUNCTION GIRDEROK(trialid)
      BEGIN
        ONC'NOT selectok (girderok := 'false',
                                ERRORC'data missing for girderok").
```

```
RETURN)")

    DBMS("SELECT(conceptok, beamok, stiffok, fspliceok, wspliceok from
          GIRDER where alternative = trialid, selectok)")

    girderok := conceptok AND beamok AND stiffok AND fspliceok AND
          wspliceok
  END
```

5.2.3. CONCEPTUAL DESIGN PHASE

5.2.3.1. ALGORITHM

The following steps are included in the conceptual design phase implementation.

1. Activate constraints on existing data that is retrieved by this phase. If the activate is not successful, return.

2. Repeat the following process until the web dimensions are successfully designed and inserted into the database.

   a. Choose the web dimensions.

   b. In a single DBMS transaction, activate the constraints on the web slenderness (conhtok) and the moment of inertia of the web (coniok) and insert the web dimensions.

3. Repeat the following process for the positive and negative moment sections until the flange dimensions are successfully designed and inserted into the database.

   a. Choose the flange dimensions.

   b. In a single DBMS transaction, activate the constraints on the clearance (clearok), the moment of inertia of the flange (coniflangeok), the flange dimensions (conflangeok) and the change in section heights (changeok) and insert the flange dimensions.

The following Pascal-like procedure can be used to perform the conceptual design phase.

```
PROCEDURE CONCEPT(trialid)
  BEGIN
    DBMS("ACTIVATE(lengthok, connok, gradeok where alternative =
          trialid, activateok)")
```

```
        IF NOT activateok THEN
            RETURN

{ CHOOSE WEB DIMENSIONS }
        REPEAT
            REPEAT
                SIZEHTW(h, tw, quit)
                IF quit THEN
                    RETURN


                DBMS("ACTIVATE(conhtok, coniok where alternative =
                                trialid, activateok)
                        INSERTUnto WSECTIONSOi, tw): <h, tw> where
                            alternative = trialid, insertok)")
            UNTIL insertok

{ CHOOSE FLANGE DIMENSIONS }
            FOR j ™ 1 TO 2
              BEGIN
                IF (j = 1) THEN
                    pmom :* 'false'
                ELSE
                    pmom := 'true'

                REPEAT
                    SIZEBFTF<bf, tf, quit)
                    IF NOT quit THEN
                          IF j = 2 THEN
                              DBMS("ACTIVATE(changeok where alternative =
                                          trialid, activateok)")
                          DBMS("ACTIVATE{clearok, coniflangeok, conflangeok
                                      where alternative = trialid,
                                      activateok)
                                  INSERTUnto FSECTIONS: <trialid, pmom,
                                      bf, tf>, insertok)")
                UNTIL insertok OR quit

              END
            DBMS("INVOKE{conceptok where alternative = trialid, invokeok)
            IF NOT invokeok THEN
                DBMS("DEACTIVATE{conhtok, coniok. coniflangeok,
                            conflangeok)")
          UNTIL invokeok
          DBMS("ACTIVATE(conceptok where alternative = trialid, activateok)")
        END
```

5.2.4. BEAM SIZING PHASE

5.2.4.1. ALGORITHM

The following steps are included in the beam sizing design phase implementation.

1. Activate constraints on existing data that is retrieved by this phase. If the activate is not successful, return.

2. Perform the analysis.

3. In a single DBMS transaction, invoke the constraint on the web slenderness (htok) and select its value. If htok is satisfied, activate it. If it is not satisfied, repeat the following steps until all web dimensions are satisfactory (htok is satisfied).

   a. Choose new web dimensions.

   b. In a single DBMS transaction, activate htok and update with the new web dimensions.

4. In a single DBMS transaction, invoke the bending stress constraint (stressok) and select the value of the logical variable for the type of moment section (posmom) for the sections with a stressok equal to false.

5. If stressok equals true for both sections, activate stressok. If stressok equals false for either section, repeat the following steps for all sections where the bending stress constraint is violated until a new set of flange dimensions are selected and the constraint is satisfied.

   a. Choose new flange dimensions.

   b. In a single DBMS transaction, activate stressok and insert the new values.

6. In a single DBMS transaction, invoke the deflection limit constraint (defok) and select the values of logical variable for the moment type (posmom) for the sections with defok equal to false.

7. If defok is true for all sections, activate defok. Repeat the following steps for all sections where the deflection limit constraint is violated until a new set of dimensions are selected.

   a. Choose new web dimensions.

   b. Update with new values.

   c. Choose new flange dimensions.

d. Update with new values.

8. Invoke the constraint on the change in section dimensions (iok).

9. In a single DBMS transaction, invoke the constraint on the overall beam sizing phase (beamok) and select its value. If beamok equals false, and the user has not failed to select new dimensions, select new section locations based on zero moment points from the analysis, deactivate htok, stressok, defok and iok and repeat the entire phase again.

The following Pascal-like procedure can be used to perform the beam sizing phase.

```
PROCEDURE BEAMSIZE(trialid)
  BEGIN

    DBMSC'ACTIVATEdengthok, connok, clearok, changeok, supportlocok
                  where alternative = trialid, activateok)")
    IF NOT activateok THEN
       RETURN
    ONC'NOT selectok (ERRORC'data missing for beam sizing phase"),
                              RETURN)")

    REPEAT

       reanalyze := 'true'
       ANALYZE(trialid)

{ CHECK WEB DIMENSIONS }
       DBMSC'INVOKEOitok where alternative = trialid, invokeok)")

       IF NOT invokeok THEN
         BEGIN
           REPEAT
              SIZEHTW(h, tw, quit)
              IF NOT quit THEN
                 DBMS("ACTIVATE(htok where alternative = trialid,
                                activateok)
                       UPDATE<WSECTIONS(h, tw): <h, tw> where
                            alternative = triaiid, updateok)")

           UNTIL updateok OR quit
           IF quit THEN
              reanalyze := 'false'
         END
       ELSE
          DBMS("ACTIVATE<htok where alternative = trialid, activateok)")
```

```
{ CHECK STRESS IN NEGATIVE AND POSITIVE MOMENT SECTION }
          DBMS("INVOKE(stressok where alternative = trialid, invokeok)
               SELECT(stressok into stress, posmom into pmom, COUNT
               into  num where alternative = trialid and stressok =
               'false', selectok)")
          IF num = 0 THEN
            DBMS("ACTIVATE(stressok where alternative = trialid,
                        activateok)")
          ELSE
            BEGIN
              FOR j := 1 TO num
                BEGIN
                  REPEAT
                      SIZEBFTF(bf, tf, quit)
                      IF quit THEN
                          reanalyze := 'false'
                      ELSE
                          DBMS("ACTIVATE(stressok where alternative =
                                       trialid, activateok)
                                UPDATE(FSECTIONS(bf, tf): <bf, tf>
                                       where posmom = pmom(j) and
                                       alternative = trialid,
                                       updateok)")
                  UNTIL updateok OR quit
                END
            END

{ CHECK THE DEFLECTION IN EACH SECTION FOR EACH SPAN }

          DBMS("INVOKE(defok where alternative = trialid, invokeok)
               SELECT(posmom into pmom, COUNT into num from
                     FSECTIONS where alternative = trialid and defok
                     = 'false', selectok)")
          IF NOT (num = 0) THEN
            BEGIN
              FOR j = 1 TO num
                BEGIN
                  REPEAT
                      SIZEHTW(h, tw, quit)
                      IF NOT quit THEN
                          DBMS("UPDATE(WSECTIONS(h, tw): <h, tw> where
                                       alternative = trialid, updateok)")
                  UNTIL updateok OR quit
                  IF quit THEN
                      reanalyze := 'false'
                  REPEAT
                      SIZEBFTF(bf, tf, quit)
                      IF NOT quit THEN
                          DBMS("UPDATE(FSECTIONS(bf, tf): <bf, tf> where
                                    posmom = pmom(j) and alternative =
                                    trialid, updateok)")
                  UNTIL updateok OR quit
```

```
                    IF quit THEN
                        reanalyze := 'false'
                    END
                END
            ELSE
                DBMS("ACTIVATE(defok where alternative = trialid, activateok)")

{ CHECK FOR LARGE CHANGE IN i }

            DBMS("ACTIVATE(iok where alternative = trialid, activateok)")

{ CHECK BEAMOK }

            DBMS<"INVOKE(beamok where alternative = trialid, invokeok)

            IF (NOT invokeok) AND (reanalyze) THEN
              BEGIN
                CALCSEGMENTS
                DBMS("DEACTIVATE(htok, stressok, defok, iok)")
              END
          UNTIL invokeok OR (NOT reanalyze)
          IF beamok THEN
            DBMS("ACTIVATE(beamok)")

        END    { BEAMSIZE }
```

5.2.5. STIFFENS* AND SPLICE DESIGN PHASE

5.2.5.1. ALGORITHM

The following steps are included in the stiffener and splice design phase implementation.

1. Activate constraints on existing data that is retrieved by the stiffener procedure. If the activate is not successful, return.

2. Activate the constraints on the stiffener height (hbsok), width (bbsok), thickness (tbsok) slenderness ratio (ccok), force (forceok), location (bslocok) and bearing capacity (bearingok).

3. For each support, repeat the following until all stiffeners are successfully designed and inserted into the database.

    a. Choose stiffener dimensions.

    b. Inset stiffener data.

    c. Activate the constraint on the overall design of the stiffeners (stiffok).

4. Activate constraints on existing data that is retrieved by the splice design procedure. If the activate is not successful, return.

5. Activate the constraints on the number of bolts (fnboltsok), the length (fslength) the remaining area (afremok), the location of the flange splice plate (fslocok), the width of the outside flange splice plate (book), the width of the inside flange splice plate (biok), the capacity of the splice plates (splicecapok) for the flange splice. Also, activate the constraints on the section modulus of the splice plate (sok), the width (bwsok), the length (wslength), the remaining area (awremok), the force per bolt for the web splice (boltfok), the moment of inertia (ispliceok) and the location of the web splice plate (wslocok) for the web splice.

6. For each splice, repeat the following process until all splices are successfully designed and inserted into the database.

   a. Select all flange splice plate dimensions.

   b. Insert the flange splice plate dimensions.

   c. Activate the overall constraint on the flange splice design (fspliceok)

   d. Choose all web splice plate dimensions.

   e. Insert the web splice plate dimensions.

   f. Activate the overall constraint on the web splice design (wspliceok).

The following procedures can be used to perform the third phase of the design process.

```
PROCEDURE STIFF(trialid)
  BEGIN
    DBMS("ACTIVATE(supportlocok, clearok, gradeok where alternative =
                   trial id, activateok)")
    IF NOT activateok THEN
      RETURN

    ONC'NOT selectok (ERRORC'data missing for stiffener phase"),
                              RETURN)")


{ CHOOSE STIFFENER DIMENSIONS }

    DBMS("SELECT(supportloc, COUNT into num from SUPPORTS order by
                 supportloc where alternative = trial id, selectok)")
```

```
    DBMS("ACTIVATE(tbsok, ccok, forceok, bearingok, bbsok, hbsok,
                   bslocok where alternative = trialid, activateok)")

    hbs := h
    FOR j := 1 to num DO
      BEGIN
        REPEAT
            SIZETBSBBS(tbs, bbs, quit)
            IF quit THEN
                RETURN

            ELSE
                DBMS("INSERT(into BSTIFFENERS(alternative, bsloc,
                             hbs, tbs, bbs): <trialid, supportloc(j),
                             hbs, tbs, bbs>, insertok)")
        UNTIL insertok
      END
    DBMS("ACTIVATE(stiffok where alternative = trialid, activateok)")

  END    { STIFF }

PROCEDURE SPLICES(trialid)
  BEGIN

    DBMSC'ACTIVATEOengthok, connok where alternative = trialid,
                  activateok)")
    IF NOT activateok THEN
        RETURN

    ONC'NOT selectok (ERRORC'data missing for splice phase"),
                           RETURN)")

    DBMS("SELECT(sectionid into sid, slend, COUNT into num from
                 SEGMENTS order by slend where alternative = trialid,
                 selectok)")

    DBMS("ACTIVATE(afremok, fslengthok, biok, book, fslocok, fslengthok,
                   splicecapok, fnboltsok, sok, awremok, boltfok,
                   ispliceok, bwsok, wslengthok, insertok, wslocok
                   where alternative = trialid, activateok)")

    FOR j := 2 TO num DO
      BEGIN
        resize := 'true'
        REPEAT
            SIZEFNL(fnlines, fslength, fnbolts. quit)
            SIZEBOTO(bo, to, bi, ti, quit)
            DBMSC'INSERTOnto FSPLICES: <trialid, slend(j), bo, to,
                          bi, ti, fslength, fnlines, fnbolts>,
                          insertok)")
        UNTIL insertok OR quit
```

```
        IF NOT quit THEN
            DBMS("ACTIVATE(fspliceok where alternative = trialid,
                        activateok)")

        REPEAT

            SIZEBWSTWS(bws, tws, quit)
            SIZEWNL(wslength, wnlines, wnbolts, quit)
            IF NOT quit
                DBMS("INSERT(into WSPLICES: <trialid, slend(j),
                            bws, tws, wslength, wnlines,
                            wnbolts>, insertok)")

        UNTIL insertok OR quit

        IF NOT quit THEN
            DBMS("ACTIVATE(wspliceok where alternative = trialid,
                        activateok)")

    END

END  { SPLICES }
```

5.2.6. ALTERNATIVE DESIGN PROCEDURES

The previous section described the implementation of the design example using the same process as the normalized database example. However, as stated before, the augmented database model provides a great deal of flexibility. This section presents examples of how useful this flexibility can be.

5.2.6.1. INTERACTIVE DESIGN

In the examples in the previous sections, the design was performed by a computer program (with associated external procedures which performed specific calculations or requested user input). However, a closer look at the augmented database example shows that except for the analysis procedure, the design could be performed interactively with the database system alone. The design phases do no more than call external procedures or DBMS operations.

For example, to perform the conceptual design phase, a designer would first choose the web and flange dimensions, such as a web 50 by 1, a positive section flange 16 by 1 1/4 and a negative section flange 16 by 1 3/4. Next, the following database commands are specified:

```
ACTIVATE(conhtok, coniok)

INSERT INTO WSECTIONS(alternative, h,tw): <1, 50.0, 1.0>

ACTIVATE(clearok, coniflangeok, conflangeok)

INSERT INTO FSECTIONS(alternative, posmom, bf, tf): <1,
                'true', 16, 1.25>

INSERT INTO FSECTIONS(alternative, posmom, bf, tf): <1,
                'false', 16, 1.75>

ACTIVATE(changeok)

ACTIVATE(conceptok)
```

If any of the inserts violate any of the constraints, the database system would print an error message and the designer could select new dimensions.

The next section will show how many of the external procedures can be included in the database.

5.2.6.2. USE OF ASSIGNMENT PROCEDURES

Since the augmented database example performed the same process as the normalized database example, no assignment procedures were needed. However, the augmented database can select many of the dimensions using assignment procedures based on specific constraints. For example, the following assignment procedure can be used to choose the flange thickness using the conflangeok constraint after the flange width is chosen:

```
PROCEDURE SETTF(trialid, tf)
  BEGIN
    ON("NOT selectok (conflangeok := 'false',
                        ERROR("data missing for conflangeok"),
                        RETURN)")

    DBMS("SELECT(bf, from FSECTIONS where alternative = trialid and
                posmom = pmom,
                fy from GRADES where grade = SELECT(grade from
                STRUCTURE, selectok)")

    tf := MAX(bf/(65/(fy)^1/2), USERCHOOSES(value))
    conflangeok := 'true'
  END
```

The procedure sets the value of tf according to the minimum value allowed by the conflangeok constraint. The USERCHOOSES function is used to allow the user the option of selecting the value of tf since the conflangeok constraint is an inequality (tf must be not less than the constant specified in the constraint). Since, the SETTF procedure assigns a value to tf that satisfies the conflangeok constraint, the value of conflangeok is automatically set to 'true'.

This type of assignment procedure can be used for many of the sizing procedures by rearranging the appropriate constraints.

5.2.6.3. SPECIFICATION OF ALTERNATIVE CONSTRAINTS

Since the example assumed that no transverse stiffeners would be included in the design, both examples had to check the web slenderness constraint (conhtok) and (htok). However, if the designer decided to change the design and include transverse stiffeners, other constraints (similar to conhtok and htok) would have to be checked. In the normalized database example, significant changes would have to be made to the program to incorporate checking the new constraints. This is not true for the augmented design database. All that has to be done is to add the new constraint functions to the database and then change the program so that the correct constraints are invoked and activated. If the designer was using the database interactively with assignment procedures, the user would again just activate the appropriate constraint. Such interactive use is not possible with the normalized database.

Also, with these added constraint functions included in the database, the designer is able to "change his mind" during the design. For example, the designer could design the girders without transverse stiffeners and then if conhtok or htok are violated, he could decide to include the stiffeners (instead of changing the dimensions of the girder) and then check the new constraints. This flexibility is not included in the normalized design database example.

5.2.7. DISCUSSION OF THE AUGMENTED DESIGN DATABASE EXAMPLE

The four procedures presented above can be used to perform the three basic design phases using an augmented design database. Therefore, an overall bridge design program could call these procedures to perform the design of the girders and then continue with the design in the following manner:

```
    .
    .
    .
        .
CONCEPT(trialid)
DBMS("SELECT(conceptok from GIRDER where alternative = trialid)")
IF conceptok THEN
  BEGIN
    BEAMSIZE(trialid)
    DBMS("SELECT(beamok from GIRDER where alternative = trialid)")
    IF beamok THEN
      BEGIN
        STIFF(trialid)
        SPLICES(trialid)
      END
  END
DBMS("ACTIVATE(girderok)
      SELECT(girderok from GIRDER where alternative = trialid)")

IF girderok THEN
    .
    .

{ CONTINUE THE BRIDGE DESIGN }
    .
    .
```

The only differences between this "main" program and the one needed for the normalized database is that the design procedures do not "flag" their success or failure (since the database stores this information) and that the overall girder design constraint must be activated.

Since the database can check all types of constraints, the responsibility of insuring integrity within the database is placed on the database itself. Therefore, as long as all appropriate constraints are activated, any data item retrieved from the database is valid.

Thus, the design program is almost completely independent of the specific constraints that must be enforced. A slight change in a constraint does not affect the program. If a new constraint is added, only a small change is needed to the specific ACTIVATE command in the program so that the constraint is enforced. This independence and uncoupling of the constraint enforcement from the details of the design program provides the basis for the alternative design procedures discussed in Section 5.2.6.

This chapter has shown that a database based on the proposed augmented form can handle and enforce a wide variety of constraints and provides the necessary flexibility for engineering design.

CHAPTER SIX
SUMMARY AND CONCLUSIONS

6.1. SUMMARY

The main objective of this thesis has been to investigate the application of current and proposed constraint processing alternatives in engineering design relational database management systems. The objective has been accomplished through:

- the investigation of current constraint processing mechanisms;
- the development of a taxonomy for constraint sources and types;
- the presentation of a proposed constraint processing mechanism [19] which utilizes constraint checking functions and assignment procedures to enforce single relation-single tuple constraints;
- the extension of this proposed model to include the enforcement of all types of single and multiple relation constraints; and
- the presentation of a design example comparing the extended constraint processing mechanism to the traditional approach to constraint processing, namely, normalization.

The conclusions and recommendations for further research presented in the following sections resulted from the investigation.

6.2. CONCLUSIONS

The extended constraint processing mechanism presented herein appears to be applicable to all types of constraints which occur in engineering design. The ability to access any data from the database enables the mechanism to enforce the multiple relation constraints as well as the single relation constraints.

This mechanism can readily be incorporated into existing relational database management systems, since it fits directly into the RDBMS framework. For example, the assertion process of a RDBMS would need the following three major extensions to implement the proposed mechanism:

1. allow arbitrary mathematical and relational operations in the specification of the assertions in the query language;

2. provide a control mechanism and a query language interface to allow a user to control which constraints are to be enforced at any given time i.e. INVOKE, ACTIVATE and DEACTIVATE; and

3. include a mechanism that can process (store and retrieve the values of) the constraint status attributes for the constraint processing functions.

The efficiency of the proposed mechanism is questionable. Since most constraint functions or procedures must retrieve data from the database, it could be argued that the mechanism is to costly to implement (due to the extra processing effort). However, if the procedures are designed so that all query operations are specified at the same time, most database systems are capable of optimizing the query for efficiency. Efficiency is most likely to be a problem for multiple relation constraints, but the database schema can usually be designed to minimize the number of such constraints. Thus overall efficiency may not be a significant problem.

6.3. FUTURE WORK

The presentation of the constraint processing mechanism in this thesis has been limited to the constraint procedures themselves. However, a linkage between the database an these constraint procedures still must be developed so that all appropriate constraints are checked upon the update of a specific data item. This entails the development of a formal mechanism which can translate the assertion of the constraint procedures into an "information network" that the database can use to determine which constraints are to be checked.

Currently, the constraint functions and procedures are supplied by the database developer or user. A system similar to the one developed by Stirk [20] that can automatically generate these procedures using decision tables

based on design codes, design objectives or specific dependencies could be implemented. In addition, the techniques developed by Holtz [17] for dealing with inequality constraints may be used to develop assignment procedures for any designable quantities.

Continued work on any of the topics presented in this thesis, require a fully implemented DBMS including all of the capabilities discussed in Sections 3.4.3 and 3.4.4. Once this is completed, the design example presented in Chapter 5 can be implemented to provide an actual demonstration of the usefulness of this constraint processing mechanism.

Finally, although this mechanism has been shown to be successful for the specific design procedure described in Chapter 4, further investigations are needed to prove its applicability to other areas of engineering design.

REFERENCES

[1] Aho, A. V., Beeri, C, and Ullman, J. D.
The Theory of Joins in Relational Databases.
ACM Transactions on Database Systems 4(3):297-314, September, 1979.

[2] American Institute of Steel Construction.
Manual of Steel Construction
Eighth edition, American Institute of Steel Construction Inc., Chicago, Illinois, 1980.

[3] Bernstein, P. A., Blaustein, B. T. and Clarke, E. M.
Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data.
Proceedings Sixth International Conference on Very Large Databases , October, 1980.

[4] Blasgen, M. W., et. al.
System R: An Architectural Overview.
IBM Systems Journal 20(1):41-61, 1981.

[5] Chamberlin, D. D. and Boyce, R. F.
Sequel: A Structured English Query Language.
Proceedings of the ACM - SIGFIDET Workshop , May, 1974.

[6] Chamberlin, D. D., et. al.
Sequel 2: A Unified Approach to Data Definition, Manipulation, and Control.
IBM Journal of Research and Development 20(6):560-575, November, 1976.

[7] Codd, E. F.
Courant Computer Science Symposia Series. Volume 6: *Further Normalization of the Data Base Relational Model.*
Prentice-Hall, Englewood Cliffs, NJ, 1972, .

[8] Codd, E. F.
Relational Database: A Practical Foundation for Productivity.
Communications of the ACM 25(2k109-117, February, 1982.

[9] Date, C. J.
An Introduction to Database Systems.
Addison-Wesley, Reading, Massachusetts, 1981.
Third Edition.

[10] Fagin, R.
Multivalued Dependencies and a New Normal Form For Relational Databases.
IBM Research Report TRJ 1812 . July, 1976.

[11] Fagin, Roland.
A Normal Form for Relational Databases That Is Based on Domains and Keys.
ACM Transactions on Database Systems 6(3), September, 1981.

[12] Fagin, Ronald, Mendelzon, Alberto O., Ullman, Jeffrey D.
A Simplified Universal Relation Assumption and Its Properties.
ACM Transactions on Database Systems 7(3):343-360, September, 1982.

[13] Fenves, S. J.
Tabular Decision Logic for Structural Design.
Journal of the Structural Division of the American Society of Civil Engineers (ASCE) 92(ST6):473-490, December, 1966.

[14] Fenves, S. J., Gaylord, E. H., and Goel, S. K.
Decision Table Formulation of the 1969 AISC Specification.
Civil Engineering Studies, Structural Research Series 347, Department of Civil Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois, August, 1969.

[15] Fenves, S. J. and Rasdorf, W. J.
Treatment of Engineering Design Constraints in a Relational Database. 1982.

[16] Goel, S. K., and Fenves S. J.
Computer-Aided Processing of Design Specification.
Journal of the Structural Division of the American Society of Civil Engineers (ASCE) 97(ST1):463-479, January, 1971.

[17] Holtz, N. M.
Symbolic Manipulation of the Design Constraints: An Aid to Consistency Management.
PhD thesis, Carnegie-Mellon University, February, 1982.

[18] Honeyman, Peter.
Testing Satisfaction of Functional Dependencies.
Journal of the Association for Computing Machinery 29(3):668-667, July, 1982.

[19] Rasdorf, W. J.
Structure and Integrity of a Structural Engineering Design Database.
PhD thesis, Carnegie-Mellon University, April, 1982.

[20] Stirk, J.
Two Software Aids For Design Specification Use.
Master's thesis, Carnegie-Mellon University, October, 1981.

[21] Stonebraker, M.
Implementation of Integrity Constraints and Views by Query Modification.
Proceedings 1975 ACM-SIGMOD Conference :65-78, 1975.