CHAPTER 4

CONSTRAINTS ON GRAPHICS DATABASE

4.1 Introduction

Constraints play a very important role in computer graphics specially in supporting new declarative style of graphics programming [Tarlton 1989]. Examples of constraints in graphics systems are: (a) restricting minimum character size for legibility; (b) endpoint of a line is same as mid point of another line; (c) size of a circle must be twice as large as size of another circle; (d) text labels must be centred in boxes; (e) colour of an icon representing rainfall in a region depends on whether the rainfall is average, above average or below average, etc.

However the idea of imposing constraints on graphical data is not new. Sutherland's Sketchpad system [Sutherland 1963] was the first to realize the importance of constraints. In this system, graphical entities such as lines, circles could be constrained by point constraints, linkage constraints or angular constraints. For example, a line could be fixed at one end by point constraint allowing the other end to move freely. The endpoints of two lines could be constrained to be joined by linkage constraints, and finally two lines could be constrained to be parallel, or intersect each other at an angle. These type of constraints enable the users to draw
a quadrilateral, impose a point constraint at one corner, impose linkage constraints on the corners, and finally put angular constraints on opposite sides to make them parallel. Hence, the users could observe the quadrilateral getting converted into a parallelogram as the constraints are satisfied.

Borning's Thinglab [Borning 1979], [Borning 1981] advanced a step further by incorporating general constraints and applying them to interactive graphics simulation. The user could not only impose point, mid point constraints on an assembly but could also move the assembly satisfying these constraints. For example, a parallelogram is constructed by adding mid point constraints to the sides of a quadrilateral and then adding four lines connecting the mid points. Interactively moving or modifying the quadrilateral moves or modifies the parallelogram accordingly.

ThingLab-II [Borning 1986], [Maloney 1989] is a direct descendent of ThingLab-I and is also implemented in Smalltalk-80. It applies constraint programming techniques to construct user interfaces. The constraints are used in a number of ways: (a) to maintain consistency between application data and information displayed on the screen. For example, the length of the arrow is proportional to the magnitude of the wind vector and the direction of arrow indicates the direction of wind. Thus, if the magnitude and direction of wind vector changes,
then accordingly the length and orientation of arrow changes. (b) to maintain consistency among multiple views of data. For example, the value of temperature is displayed as text and as height of mercury column both in Celsius and Fahrenheit. These can be seen as four views (two textual and two graphical). A change in the temperature value modifies all textual and graphical information. (c) constraints can be imposed on how some information has to be formatted on the screen. For example, if a window is divided half vertically and the lower half divided into right and left halves. Constraints are used to maintain the proportions of these sub-divisions as window is re-sized. (d) constraints can be used for animation.

In Nelson’s JUNO graphics system [Nelson 1985], constraints are allowed to be specified by selecting icons and points. Its only data object is point and the system is limited to line drawing, with a line specified by pair of points. It supports four types of constraints. These are (a) the equality of the distance between points. For example, \((x,y) \text{ CONG } (u,v)\) which means that distance from \(x\) to \(y\) equals distance from \(u\) to \(v\). (b) the parallelism of two lines between points. For example, \((x,y) \text{ PARA } (u,v)\) which means that direction from \(x\) to \(y\) is parallel to direction from \(u\) to \(v\). (c) the horizontality of two points. For example, \(\text{HOR}(x,y)\) means that direction from \(x\) to \(y\) is horizontal. (d) the verticality of two points.
For example, $VER(x,y)$ means that direction from $x$ to $y$ is vertical. The novel feature of this system is that interactive editing of any displayed image results in modification of the text of the program accordingly.

Constraints can be classified into one-way and multiway constraints. In multiway constraints, when two or more objects are attached by a constraint then modification of any object implies the modification of some or all the other objects again satisfying the constraints whereas one-way constraints allow modification of only one object so that the other objects are also modified satisfying the constraint. For example, if a line is constraint to be joined to the centers of two circles by one-way constraint, then if either circle moves the line also moves. But, if the line moves, neither of the two circles move. The constraints that connect line and circles are violated or removed. If the line and circles are connected by multiway constraints, then if either circle moves the line moves as in one-way constraints and also, when the line moves either one or both circles moves to satisfy the constraints.

Both Sketchpad and Thinglab are examples of multiway constraint-based systems. For one-way constraint-based systems we refer to [Barth 1986], [Szekely 1988]. Multiway constraints are more powerful than one-way constraints. But they are difficult to implement than one-way constraints. Also they can introduce ambiguity at
the design level. For example, suppose we have the constraint \( A-B-C=0 \), and if \( A \) changes its value, then the constraint solver has to decide whether to change \( B \) or \( C \) or both. To eliminate this ambiguity, the theory of constraints hierarchy was introduced by Borning et al [Borning 1987]. Vander Zanden [Zanden 1989] proposed a new constraint-solving algorithm to solve multiway constraints in real time at the user interface development environment. Leler's Bertrand [Leler 1988] is a constraint language that is used to solve system of constraints.

In the constraint-based systems mentioned above, constraints are used to control the layout of objects on the screen. These systems allow constraints to be imposed on the geometry of objects and on the relationship between two or more objects. But, these systems do not allow multiple output devices. Hence they do not consider constraints that are important in this framework. As these systems are not database systems they do not impose constraints when new objects are inserted in or deleted from the database, or are modified.

In our scheme [Bhalla 1991a], we allow constraints to be imposed on individual objects, between two or more objects, on a class of objects, on the relationship between window, viewport and device, and part hierarchies.
4.2 CLASSIFICATION OF CONSTRAINTS

Several attempts have been made to handle integrity constraints in the databases and classify the constraints on the basis of scope of application [Buchmann 1986], [Lyngbaek 1989], [Qian 1986]. However, most of these attempts [Lyngbaek 1989], [Qian 1986] have been made in the context of tuple or record oriented database systems applicable to conventional applications.

In this chapter, we discuss constraints in the context of object-oriented graphics database systems [Bhalla 1991a]. We first distinguish between class constraints and instance constraints. We define constraints imposed on all instances of a class as class constraints and constraints imposed on some instances of a class as instance constraints. For example, in the class LINE, line objects should have distinct end points is a class constraint and in addition a line is 'd' distance from a point P is an instance constraint.

It is obvious that the class constraints can be anticipated by the database designer, but instance constraints are specified as and when the need arises. Many efforts have been made to handle class constraints in the databases [Lyngbaek 1989], [Qian1986], whereas little efforts have been made so far, to specify constraints on some instances of a class [Buchmann 1986]. We first discuss class constraints then instance constraints and
finally constraints on part hierarchy.

Before discussing these constraints, we introduce data operations [Dittrich 1988] which will be used in this chapter. To read and write values of instance variables the functions are attached with tags 'get' and 'put' respectively, to the instance variable names. For example, the function "get-xlength(r)" returns the xlength of the object r. Similarly, the function "put-name(d,'laser')" sets the value of instance variable "name" of object d to 'laser'.

The value of an attribute can be single valued or set valued. We have used ordered set. Thus, we have provided operations that can retrieve elements from an ordered set and insert a new element as the last element of a set. Some important operations are given below:

- **same(x,y)** returns a boolean which is true if the objects x and y are identical.
- **equal(x,y)** returns a boolean which is true if the objects x and y are equal.
- **member(x,S)** returns a boolean which is true if x is an element of set S.
- **empty(S)** returns a boolean which is true if the set S is empty.
- **count(S)** returns the number of elements in the set S.
- **element(i,S)** returns the ith element of set S.
- **insert(x,S)** inserts x as the last element in S.
remove(x,S) deletes element x from S.
difference(S,T) returns the difference of S and T.

4.2.1 Class Constraints

Class constraints apply to all the instances of a class. We have further classified these into domain constraints, intra-object constraints, set object bound constraints, association constraints, inter-object constraints and referential integrity constraints. We discuss these one by one.

4.2.1.1 Domain Constraints

These are constraints that apply to the domain of an instance variable of a class. The following example specifies the radius of the circle to be greater than 0.

\[
\{ c \in CIRCLE \mid \text{get-radius(c)} > 0 \}
\]

Another example of domain constraint is the specification of range, which the values of an instance variable can take, by an inequality.

\[
\{ r \in ARROW \mid 0.0 \leq \text{get-direction(r)} \leq 360.0 \}
\]

4.2.1.2 Intra-Object Constraints

Intra-object constraints involve two or more instance variables of an object. The following constraint specifies that all LINE objects must have distinct head and tail.

\[
\{ l \in LINE \mid \text{NOT(equal(get-tail(l), get-head(l)))} = \text{true} \}
\]
4.2.1.3 Set Object Bound Constraints

Set object bound constraints specify the minimum and maximum bound on the number of elements of a set valued instance variable. For example, a WINDOW object may have a minimum 0 and maximum 10 destinations. That is, a window may not be mapped to a viewport and at the most to 10 viewports.

\[ \{ w \in \text{WINDOW} \mid 0 \leq \text{COUNT}(\text{get-destination}(w)) \leq 10 \} \]

Similarly, a viewport must be mapped by at least a window and at the most by 6 windows.

\[ \{ v \in \text{VIEWPORT} \mid 0 < \text{COUNT}(\text{get-origin}(v)) \leq 6 \} \]

4.2.1.4 Association Constraints

Association constraints involve set-valued instance variables of two classes, where the values of the instance variable are references to instances of the other class and vice-versa. Association constraints are two-way constraints.

\[ \{ w \in \text{WINDOW}, v \in \text{VIEWPORT} \mid \\
\quad \text{member}(v, \text{destination}(w)) = \text{true} \\
\quad \Leftrightarrow \text{member}(w, \text{origin}(v)) = \text{true} \} \]

4.2.1.5 Inter-Object Constraints

These are constraints that involve two or more instance variables of different classes. An example of an inter-object constraint is that the size of viewport must lie within the maximum allowable size of the corresponding
device.

\[
\{ v \in \text{VIEWPORT} , \exists d \in \text{DEVICE} \mid \\
\quad 0 < (\text{get-xcor}(\text{get-location}(v)) + \text{get-xsize}(v)) \\
\quad \leq \text{get-xmaxsize}(d) \\
\text{AND} \\
\quad 0 < (\text{get-ycor}(\text{get-location}(v)) + \text{get-ysize}(v)) \\
\quad \leq \text{get-ymaxsize}(d) \}
\]

4.2.1.6 Referential Integrity Constraints

A referential integrity constraint restricts the values that an instance variable of a class can take, where the values are references to instances of another class. In object-oriented databases, when an instance of a class is deleted, the referential integrity constraint is violated if this instance is the only instance being referred by an object for which it is a required parameter.

Three approaches may be used to enforce the constraint in this situation. In the first approach, the deletion of the instance can be aborted. For example, a viewport object cannot exist without being mapped by a window. Therefore, referential integrity constraint is violated if we attempt to delete a window object, when this is the only window being mapped onto a viewport. Deletion of window in such case is aborted.

\[
\{ v \in \text{VIEWPORT} , \exists w \in \text{WINDOW} \mid \\
\quad \text{member}(w, \text{origin}(v)) = \text{true} \}
\]
In the second case, all the corresponding objects that refer to this instance are also deleted. For example, if a device is deleted, all viewport objects that refer to it can be deleted.

\[ \forall v \in \text{VIEWPORT}, \exists d \in \text{DEVICE} \mid \text{same}(d, \text{device}(v)) = \text{true} \]

In the third approach, the corresponding reference value of the object referring to it can be set to null (or deleted from the set, if the reference instance variable is set-valued). For example, if a viewport instance is deleted, then reference to this instance is removed from the set of the destinations of the window.

### 4.2.2 Instance Constraints

Instance constraints apply to one or more instances of a class. In order to specify instance constraints we treat constraints as objects. Instance constraints are two-way constraints. In this section we discuss some types of instance constraints.

#### 4.2.2.1 Distance constraint between two points

**Class PP$distance**

\[ \begin{align*}
\text{P1} & : \text{POINT} \\
\text{P2} & : \text{POINT} \\
\$\text{distance} & : \text{real}
\end{align*} \]

The objects belonging to class PP$distance have 'distance' constraint imposed between the two points. A "$\$$" sign before an attribute indicates that the value of
the attribute is calculated when the constraint object is created. The user has to specify the two points only and the distance 'd' between the points is calculated and is stored in the database.

If a point, say $P_1$, is modified such that the distance between the two points is equal to 'd' then the constraint is satisfied. But, if the distance is not equal to 'd' we modify $P_2$ so as to satisfy the constraint. The steps involved to determine $P_2$ and the results are given below. For the details of the derivation we refer to Appendix B.

Let $P_1 = P_1 (x_1, y_1)$ and $P_2 = P_2 (x_2, y_2)$ be the given points and let $d = d(P_1, P_2)$ be the distance between them. If $P_1$ is modified to $P_1'$ and $d(P_1', P_2) \neq d(P_1, P_2)$, then we modify $P_2$ to $P_2'$ (Figure 4.1) such that

(a) $P_1'P_2'$ is parallel to $P_1P_2$.
(b) distance between $P_1', P_2'$ is equal to $d$.

The algorithm is as follows:

Step 1: Determine $d(P_1', P_2)$. If $d(P_1', P_2) = d$ is true then the constraint is satisfied else go to step 2.

Step 2: Consider line $L'$ passing through $P_1'$ and parallel to $P_1P_2$.

Step 3: Define $P_2'$ lying on $L'$ having distance "d" from $P_1'$ and in the direction of $P_1P_2$. (This is to avoid ambiguity.)

The coordinates of the point $P_2' (x_2', y_2')$ are given below:

\[
\begin{align*}
x_2' &= x_1' + (x_2 - x_1) \\
y_2' &= y_1' + (y_2 - y_1)
\end{align*}
\]
FIG. 4.1: DISTANCE CONSTRAINT BETWEEN TWO POINTS.
4.2.2.2 Distance constraint between point and line

Class PL\$distance
P : POINT
L : LINE
\$distance : real

The constraint restricts the distance between a point and a line to be equal to "d". If the point is modified such that the distance between the point and line is not equal to "d", then we modify the line so as to satisfy the constraint. Similarly, if the line is modified such that the constraint is violated, then we modify the point so as to satisfy the constraint. The steps involved to modify the object (point or line) are given below. The details of the derivation are given in Appendix C and for the orientation of a triangle we refer to Appendix A.

Let \( L=L(P_1,P_2) \) and \( P=(X,Y) \) be the given line and point respectively, where \( P_1=P_1(x_1,y_1) \) and \( P_2=P_2(x_2,y_2) \).
Assume that point \( P \) does not lie on \( L \). Let \( Q=Q(x_o,y_o) \) be a point on \( L \) obtained by drawing a perpendicular from \( P \) to \( L \). Let \( d=d(P,L)=d(P,Q) \) be the perpendicular distance between the point and the line, where \( d(.,.,.) \) is the distance function.

Case 1: Assume that the line \( L \) is updated to \( L'=L'(P'_1,P'_2) \) where \( P'_1=P'_1(x'_1,y'_1) \) and \( P'_2=P'_2(x'_2,y'_2) \). If the distance between \( P \) and \( L' \) is not equal to 'd' then we modify \( P \) to \( P' \) (Figure 4.2b). To determine \( P' \) the following algorithm is used.
Step 1: Find the distance \( d(P,L') \). If \( d(P,L')=d(P,L) \)
FIG. 4.2: DISTANCE CONSTRAINT BETWEEN A POINT AND A LINE.
is true then the constraint is satisfied else go to step 2.

Step 2: Determine $Q$ lying on $L(P_1, P_2)$ such that

$$\overrightarrow{PQ} \cdot \overrightarrow{P_1P_2} = 0.$$ 

Step 3: Find $Q'$ as the transformed point of $Q$ and lying on $L'$.

Step 4: Find $P'$ such that

(a) $\overrightarrow{P'Q'} \cdot \overrightarrow{P_1P_2} = 0$

(b) $d(P', Q') = d(P, Q)$

(c) Triangles $\Delta PP_1P_2$ and $\Delta P'P_1P_2'$ have same orientation.

If triangle $\Delta PP_1P_2$ is clockwise oriented, then $P'(X', Y')$ is calculated as given below:

$$X' = x_1' + u_o (x_2' - x_1') + \frac{(y_2' - y_1')}{(x_2' - x_1')^2 + (y_2' - y_1')^2} \frac{d}{\sqrt{(x_2' - x_1')^2 + (y_2' - y_1')^2}}$$

and

$$Y' = y_1' + u_o (y_2' - y_1') - \frac{(x_2' - x_1')}{(x_2' - x_1')^2 + (y_2' - y_1')^2} \frac{d}{\sqrt{(x_2' - x_1')^2 + (y_2' - y_1')^2}}$$

where

$$d = d(P, L) = \frac{|(x_1 - X)(y_2 - Y) - (y_1 - Y)(x_2 - x_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

$$u_o = \frac{-(x_1 - X)(x_2 - x_1) + (y_1 - Y)(y_2 - y_1)}{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

If triangle $\Delta PP_1P_2$ is counter clockwise oriented, then $P'(X', Y')$ is calculated as given below:

$$X' = x_1' + u_o (x_2' - x_1') - \frac{(y_2' - y_1')}{(x_2' - x_1')^2 + (y_2' - y_1')^2} \frac{d}{\sqrt{(x_2' - x_1')^2 + (y_2' - y_1')^2}}$$

and
\[ Y' = y'_1 + u_0 (y'_2 - y'_1) + \frac{(x'_2 - x'_1) \, d}{[(x'_2 - x'_1)^2 + (y'_2 - y'_1)^2]^{1/2}} \]

where \( u \) and \( d \) are as given above.

**Case 2:** Assume that the point \( P \) is updated to \( P' = P'(X', Y') \). If the distance between \( P' \) and \( L \) is not equal to \( 'd' \) then we modify \( L \) to \( L' \) (Figure 4.2c). To determine \( L' \) we use the following algorithm.

Step 1: Find the distance \( d(P', L) \). If \( d(P', L) = d(P, L) \) is true then the constraint is satisfied else go to step 2.

Step 2: Determine \( Q \) lying on \( L(P_1, P_2) \) such that \( \overrightarrow{PQ} \cdot \overrightarrow{P_1P_2} = 0 \).

Step 3: Consider line \( L' \) passing through \( P' \) and parallel to \( \overrightarrow{PQ} \).

Step 4: Find \( Q' \) as the transformed point of \( Q \) lying on \( L' \) and in the direction of \( PQ \) having a distance "\( d' \)" from \( P' \).

Step 5: Define a new line \( L' \) passing through \( Q' \) and parallel to \( L \).

Step 6: Define \( P'_1 \) and \( P'_2 \) on \( L' \) and in the direction of \( \overrightarrow{QP_1} \) and \( \overrightarrow{QP_2} \) respectively such that \( d(P'_1, Q) = d(P'_1, Q') \) and \( d(P'_2, Q) = d(P'_2, Q') \).

The line \( L'(P'_1, P'_2) \) where points \( P'_1(x'_1, y'_1) \) and \( P'_2(x'_2, y'_2) \) are defined by

\[
\begin{align*}
x'_1 &= X' + (x_1 - X) \\
y'_1 &= Y' + (y_1 - Y) \\
x'_2 &= X' + (x_2 - X) \\
y'_2 &= Y' + (y_2 - Y)
\end{align*}
\]
4.2.2.3 Two lines constrained to be parallel

Class LL$\parallel$LINE
L1 : LINE
L2 : LINE

This constraint restricts two lines to be parallel always. If a line is modified in such a way that it remains parallel to the other line, then the constraint is satisfied. But if one line is updated in such manner that the two lines are no longer parallel, then we modify the other line so as to satisfy the constraint. The steps involved to modify the other line and the results are given below. The details of the derivation are given in Appendix D.

Let $L_1=L_1(P_1,P_2)$ and $L_2=L_2(Q_1,Q_2)$ be the two given lines, where $P_1=(x_1,y_1)$, $P_2=(x_2,y_2)$, $Q_1=(X_1,Y_1)$ and $Q_2=(X_2,Y_2)$. Suppose the line $L_1$ is modified to $L'_1(P'_1,P'_2)$ where $P'_1=(x'_1,y'_1)$ and $P'_2=(x'_2,y'_2)$. If the lines $L'_1$ and $L_2$ are not parallel, then we modify $L_2$ to $L'_2(Q'_1,Q'_2)$ (Figure 4.3) where $Q'_1=(X'_1,Y'_1)$ and $Q'_2=(X'_2,Y'_2)$ such that

(i) $L'_2$ is parallel to $L'_1$.

(ii) $d(L_1,L_2)=d(L'_1,L'_2)=d$

The steps followed are given below:

Step 1: Find the distance "d" between lines $L_1$ and $L_2$.

Step 2: If $d>0$, go to step 3. If $d=0$, find $Q'$ as the transformed point of $Q_1$ and lying on the line determined by $P'_1$ and $P_2$, and go to step 8.

Step 3: If $L'_1$ and $L_2$ are parallel then the constraint
Fig. 4.3: Two lines constrained to be parallel.

(a) Original lines $L_1$ and $L_2$.

(b) $L_1$ is moved to $L'_1$ and so $L_2$ is moved into $L'_2$ (or vice-versa).
satisfied else go to step 4.

Step 4: Find the point $P$ lying on $L_1$ such that
\[
\overrightarrow{PQ_1} \cdot \overrightarrow{P_1P_2} = 0.
\]

Step 5: Find the point $P'$ lying on $L'_1$ as the transformed point of $P$.

Step 6: Find $Q'$ such that
(a) $\overrightarrow{P'Q'} \cdot \overrightarrow{P_1'P_2'} = 0$.
(b) $d(P',Q'_1) = d(P,Q_1)$
(c) triangles $\Delta P_1P_2Q_1$ and $\Delta P'_1P'_2Q'_1$ have same orientation.

Step 7: Find the line $L'_2$ passing through $Q'_1$ and parallel to $\overrightarrow{P_1'P_2'}$.

Step 8: Define $Q'_2$ lying on $L'_2$ such that
\[
\overrightarrow{Q'_1Q'_2} = \alpha \overrightarrow{P_1'P_2'}
\]
where $\alpha$ satisfies
\[
\overrightarrow{Q_1Q_2} = \alpha \overrightarrow{P_1P_2}.
\]

The point $Q'_1(X'_1,Y'_1)$ is defined as either
\[
X'_1 = x' + \frac{(y'_2 - y'_1) \cdot d}{d(P'_1,P'_2)},
\]
\[
Y'_1 = y' - \frac{(x'_2 - x'_1) \cdot d}{d(P'_1,P'_2)}
\]
or
\[
X'_1 = x' - \frac{(y'_2 - y'_1) \cdot d}{d(P'_1,P'_2)},
\]
\[
Y'_1 = y' + \frac{(x'_2 - x'_1) \cdot d}{d(P'_1,P'_2)}
\]
such that triangles $\Delta P_1P_2Q_1$ and $\Delta P'_1P'_2Q'_1$ have the same orientation.

Knowing $Q'_1(X'_1,Y'_1)$, the other point $Q'_2(X'_2,Y'_2)$ is defined as
\[
x'_2 = x'_1 + \frac{(x_2 - x_1)}{(x_2 - x_1)} (x'_2 - x'_1),
\]

\[
y'_2 = y'_1 + \frac{(x_2 - x_1)}{(x_2 - x_1)} (y'_2 - y'_1)
\]

if \((x_2 - x_1) \neq 0\).

and

\[
x'_2 = x'_1 + \frac{(y_2 - y_1)}{(y_2 - y_1)} (x'_2 - x'_1),
\]

\[
y'_2 = y'_1 + \frac{(y_2 - y_1)}{(y_2 - y_1)} (y'_2 - y'_1)
\]

if \((x_2 - x_1) = 0\).

**Case 2:** Similarly if \(L_2\) is moved to \(L'_2\) and suppose that the constraint is not satisfied then we move \(L_1\) to \(L'_1\) so as to satisfy the constraint. This can be deduced by interchanging \(L_1\) and \(L_2\).

**4.2.2.4 Angle Constraint Between Two Lines**

**Class LL$\angle$angle**

\(L1\) : LINE

\(L2\) : LINE

$\angle$angle : real

This constraint restricts the angle between the two lines to be equal to "\(\theta\)". If a line is modified in such a way that it makes an angle "\(\theta\)" with the other line, then the constraint is satisfied. Otherwise, we modify the other line so as to satisfy the constraint. The steps involved to modify the other line and the results are given below. The details of the derivation are given in Appendix E.
Step 1: Find the point of intersection $P$ of lines $L_1(P_1, P_2)$ and $L_2(Q_1, Q_2)$.

Step 2: Find the angle $\theta$ between vectors $P_1P_2$ and $Q_1Q_2$.

Step 3: Find the point of intersection $P'$ of lines $L_1'$ and $L_2$.

Step 4: Find the angle $\psi$ between vectors $P_1'P_2'$ and $Q_1Q_2$.

Step 5: If $\psi = \theta$ or $\psi = 180 - \theta$ go to step 6 else go to step 12.

Step 6: If the point of intersection $P'$ of the lines passing through $P_1'P_2'$ and $Q_1Q_2$ lie on the line segments $P_1'P_2'$ and $Q_1Q_2$, then the constraint is satisfied. If $\psi = 180 - \theta$, then define $Q_1' = Q_2$ and $Q_2' = Q_1$.

Step 7: If $P'$ lies on line segment $L_1'$ but does not lie on $Q_1Q_2$, then we move $Q_1Q_2$ to $Q_1'Q_2'$ (Figure 4.4a) such that $|PQ_1|:|Q_1Q_2| = |P'Q_1'|:|Q_1'Q_2'|$.

$Q_1'(X_1', Y_1')$ and $Q_2'(X_2', Y_2')$ are as given below:

$X_1' = X_1 + (v_1 - u_1)(X_2 - X_1)$,

$Y_1' = Y_1 + (v_1 - u_1)(Y_2 - Y_1)$,

and

$X_2' = X_1 + (1 + v_1 - u_1)(X_2 - X_1)$,

$Y_2' = Y_1 + (1 + v_1 - u_1)(Y_2 - Y_1)$,

where

$u_1 = \frac{(Y_1 - Y_2')(X_2 - X_1) - (X_1 - X_2')(Y_2 - Y_1)}{(X_2 - X_1)(Y_2 - Y_1) - (Y_2 - Y_1)(X_2 - X_1)}$,

$v_1 = \frac{(Y_1 - Y_2')(X_2' - X_1') - (X_1 - X_2')(Y_2' - Y_1')}{(X_2 - X_1)(Y_2 - Y_1) - (Y_2 - Y_1)(X_2 - X_1)}$,

and $P_1 = P_1(x_1, y_1), P_2 = P_2(x_2, y_2), Q_1 = Q_1(x_1, y_1)$. 


(a) $\cos \psi = \pm \cos \theta$ AND $P'$ LIES ON $L_1$ BUT NOT ON $L_2$

(b) $\cos \psi = \pm \cos \theta$ AND $P'$ LIES ON $L_2$ BUT NOT ON $L_1$

FIG. 4.4: ANGLE CONSTRAINT BETWEEN TWO LINES.
Q_2 = Q_2(X_2, Y_2), P'_1 = P'_1(x'_1, y'_1), P'_2 = P'_2(x'_2, y'_2),
Q'_1 = Q'_1(X'_1, Y'_1), Q'_2 = Q'_2(X'_2, Y'_2).

Go to step 11.

Step 8: If P' does not lie on line segment P'_1P'_2 find the transformed point P'' of P on the line P'_1P'_2.

Step 9: If P' lies on Q_1Q_2' but does not lie on the line segment P'_1P'_2 then find points Q'_1 and Q'_2 (Figure 4.4b) lying on the line passing through P'' and parallel to Q_1Q_2 such that

|Q_1P'|:|Q_1Q_2'| = |Q'_1P''|:|Q'_1Q'_2'|.

Q'_1 = Q'_1(X'_1, Y'_1) and Q'_2 = Q'_2(X'_2, Y'_2) are as given below:

X'_1 = x'_1 + u_o(x'_2 - x'_1) + (-v_1)(x_2 - x_1),
Y'_1 = y'_1 + u_o(y'_2 - y'_1) + (-v_1)(y_2 - y_1),

and

X'_2 = x'_1 + u_o(x'_2 - x'_1) + (1 - v_1)(x_2 - x_1),
Y'_2 = y'_1 + u_o(y'_2 - y'_1) + (1 - v_1)(y_2 - y_1),

where

u_o = \frac{(x_2 - x_1)(Y_1 - Y_1) - (x_1 - x_1)(Y_2 - Y_1)}{(x_2 - x_1)(y_2 - y_1) - (y_2 - y_1)(x_2 - x_1)},

v_1, (x_i, y_i), (X_i, Y_i) and (x'_i, y'_1) are defined as above.

Go to step 11.

Step 10: If P' does not lie on Q_1Q_2', we determine points Q'_1 and Q'_2 (Figure 4.4c) lying on line passing through P'' and parallel to Q_1Q_2 such that

|Q_1P|:|Q_1Q_2| = |Q'_1P''|:|Q'_1Q'_2|. Define

Q'_1 = Q'_1(X'_1, Y'_1) and Q'_2 = Q'_2(X'_2, Y'_2) as given below:

X'_1 = x'_1 + u_o(x'_2 - x'_1) + (-u_1)(x_2 - x_1),
(C) $\cos \psi = \pm \cos \theta$ AND $P'$ DOES NOT LIE ON BOTH $L'_1$ AND $L_2$

(d) $\cos \theta \neq \pm \cos \psi$

FIG. 4.4 (cont.): ANGLE CONSTRAINT BETWEEN TWO LINES.
\[ y'_1 = y'_1 + u_0(y'_2 - y'_1) + (-u_1)(y'_2 - y'_1), \]
and
\[ x'_2 = x'_1 + u_0(x'_2 - x'_1) + (1 - u_1)(x'_2 - x'_1), \]
\[ y'_2 = y'_1 + u_0(y'_2 - y'_1) + (1 - u_1)(y'_2 - y'_1), \]
where \((x_i, y_i), (x'_i, y'_i), (X_i', Y_i')\) and \(u_0, u_1\)
are as defined above.

Step 11: If \(\psi = 180 - \Theta\) then interchange \(Q'_1\) and \(Q'_2\).

Step 12: If \(\psi \neq \Theta\) and \(\psi \neq 180 - \Theta\), find the transformed
point \(P'\) of \(P\) on the line segment \(P'_1P'_2\).

Step 13: Find the orientation of triangle \(\Delta P_1PQ_1\).

Step 14: If the orientation of triangle \(\Delta P_1PQ_1\) is counter-
clockwise (clockwise), then we determine line \(L'\)
by rotating vector \(P'P_1\) about the point \(P'\)
clockwise (counter-clockwise) by angle \(\Theta\).

Step 15: Determine points \(Q'_1\) and \(Q'_2\) (Figure 4.4d) lying
on \(L'\) such that \(|Q_1P|:|Q_1Q_2| = |Q'_1P'|:|Q'_1Q'_2|\).
We define \(Q'_1 = Q'_1(X'_1, Y'_1)\) and \(Q'_2 = Q'_2(X'_2, Y'_2)\) as below:
\[ x'_1 = x' + \frac{d(Q_1, Q_2)}{d(P', P'_1)} [(x'_1 - x') \cos + (y'_1 - y') \sin \Theta] \]
\[ y'_1 = y' + \frac{d(Q_1, Q_2)}{d(P', P'_1)} [(x'_1 - x') \sin + (y'_1 - y') \cos \Theta] \]
\[ x'_2 = x' + (u_1 - 1) \frac{d(Q_1, Q_2)}{d(P', P'_1)} [(x'_1 - x') \cos + (y'_1 - y') \sin \Theta] \]
\[ y'_2 = y' + (u_1 - 1) \frac{d(Q_1, Q_2)}{d(P', P'_1)} [(x'_1 - x') \sin + (y'_1 - y') \cos \Theta] \]
when the triangle \(\Delta P_1PQ_1\) is clockwise oriented
and
\[
X'_1 = x' + u_1 \frac{d(Q_1, Q_2)}{d(P', P') \sin \theta} \cos \theta + (y'_1 - y') \sin \theta
\]
\[
Y'_1 = y' + u_1 \frac{d(Q_1, Q_2)}{d(P', P') \sin \theta} \sin \theta + (y'_1 - y') \cos \theta
\]
\[
X'_2 = x' + (u_1 - 1) \frac{d(Q_1, Q_2)}{d(P', P') \sin \theta} \cos \theta + (y'_2 - y') \sin \theta
\]
\[
Y'_2 = y' + (u_1 - 1) \frac{d(Q_1, Q_2)}{d(P', P') \sin \theta} \sin \theta + (y'_2 - y') \cos \theta
\]

when the triangle \( \Delta P_1 P Q_1 \) is counter-clockwise oriented;

where \( P'(x', y') = \left( x'_1, y'_1 \right) + u_0 (x'_2 - x'_1, y'_2 - y'_1) \);
\( d(., .) \) is the distance function; \( u_0, u_1 \) and
\( (x'_1, y'_1), (x_1, y_1), (x'_2, y'_2) \) are defined as above.

4.2.2.5 Two circles constrained to touch each other externally

Class CC$1$point
C1 : CIRCLE
C2 : CIRCLE

The constraint restricts that the two circles touch each other externally. If a circle is modified such that it touches the other circle externally then the constraint is satisfied. If one circle is modified and the constraint is violated then we modify the other circle so as to satisfy the constraint. The steps involved to modify the other circle and the results are given below. The details of derivation are given in Appendix F.
Let $C_1$ and $C_2$ be two circles with centres $C_1 = (x_1, y_1)$ and $C_2 = (x_2, y_2)$ and radii $r_1$ and $r_2$ respectively.

**Case 1:** Suppose the circle $C_1$ is modified to another circle $C'_1$ with centre $C'_1 = (x'_1, y'_1)$ and radius $r'_1$. If $C'_1$ and $C_2$ touch each other externally then the constraint is satisfied. Otherwise, we move $C_2$ to $C'_2$ (Figure 4.5) with centre $C'_2 = (x'_2, y'_2)$ and radius $r'_2$. The steps involved are as follows:

**Step 1:** Find if $C'_1$ touches $C_2$ externally. If yes then the constraint is satisfied else go to step 2.

**Step 2:** Define $r'_2$ such that

$$
\frac{r'_1}{r'_1 + r'_2} = \frac{r_1}{r_1 + r_2}
$$

**Step 3:** Consider a line $L$ passing through $C'_1$ and parallel to $C_1C_2$.

**Step 4:** Define $C'_2$ lying on $L$ and in the direction of $\overrightarrow{C_1C_2}$ such that $d(C'_1, C'_2) = r'_1 + r'_2$

The new radius $r'_2$ and the centre $C'_2(x'_2, y'_2)$ are given by:

$$
\begin{align*}
    r'_2 &= -r'_1 + \left( \frac{r'_1}{r_1} \right) (r_1 + r_2) ; \\
    x'_2 &= x'_1 + \left( \frac{r'_1}{r_1} \right) (x_2 - x_1) ; \\
    y'_2 &= y'_1 + \left( \frac{r'_1}{r_1} \right) (y_2 - y_1) .
\end{align*}
$$

**Case 2:** If circle $C_2$ is modified to $C'_2$ and the constraint is not satisfied, then we modify the circle $C_1$ to $C'_1$ such that again $C'_1$ and $C'_2$ touch externally. This case is
FIG. 4.5: TWO CIRCLES ARE CONSTRAINED TO TOUCH EACH OTHER EXTERNALLY.

FIG. 4.6: TWO CIRCLES ARE CONSTRAINED TO INTERSECT AT TWO POINTS.
exactly the same as of case 1 and in fact the results can be obtained by interchanging $c_1$ and $c_2$.

4.2.2.6 Two circles constrained to intersect each other at two points

Class CC$^2$point
C1 : CIRCLE
C2 : CIRCLE

This constraint restricts that the two circles intersect each other at two distinct points. If one circle is modified in such a way that it intersects the other circle at two points, then the constraint is satisfied. If one circle is modified but the constraint is violated then the other circle is also modified so as to satisfy the constraint. The steps involved to modify the circle and the results are given below. The details of the derivation are given in Appendix G.

Let $c_1$ and $c_2$ be two circles with centres $C_1 = C_1 (x_1', y_1')$, $C_2 = C_2 (x_2', y_2')$ and radii $r_1$, $r_2$ respectively. Case 1: Suppose the circle $c_1$ is modified to another circle $c_1'$ with centre $C_1' (x_1', y_1')$ and $r_1'$. If $c_1'$ and $c_2$ intersect at two distinct points then the constraint is satisfied. Otherwise, we move $c_2$ to $c_2'$ (Figure 4.6) with centre $C_2' (x_2', y_2')$ and radius $r_2'$ satisfying the constraint, as follows:

Step 1: Find if $c_1'$ intersects $c_2$ at two points. If yes then the constraint is satisfied else go to step 2.
Step 2: Define $r'_2$ such that
\[
\frac{r'_1}{r'_1 + r'_2} = \frac{r_1}{r_1 + r_2}
\]
Step 3: Consider a line $L$ passing through $C'_1$ and parallel to $C_1C_2$.
Step 4: Define $C'_2$ lying on $L$ and in the direction of $C_1C_2$ such that
\[
\frac{d(C'_1, C'_2)}{r'_1} = \frac{d(C_1, C_2)}{r_1}
\]
We define the new radius $r'_2$ and centre $C'_2(x'2, y'_2)$ as:
\[
r'_2 = -r'_1 + \left( \frac{r'_1}{r_1} \right) (r_1 + r_2);
\]
\[
x'_2 = x'_1 + \left( \frac{r'_1}{r_1} \right) (x_2 - x_1),
\]
\[
y'_2 = y'_1 + \left( \frac{r'_1}{r_1} \right) (y_2 - y_1).
\]
Case 2: If circle $c'_2$ is modified to $c'_2$ and the constraint is not satisfied, then we modify the circle $c'_1$ to $c'_1$ such that again $c'_1$ and $c'_2$ intersect at two points. This case is exactly the same as of case 1 and the results can be obtained by interchanging $c'_1$ and $c'_2$.

4.2.2.7 Circle constraint to be contained in another circle

Class CC$in$
Cout : CIRCLE
Cin : CIRCLE
This constraint restricts one circle to contain the other circle. If the outer circle is modified in such a way that it contains the inner circle then the constraint is satisfied. But if it is modified and the constraint is violated, then we move the inner circle so as to satisfy the constraint. Similarly, if inner circle is modified in such a way that it is contained in the outer circle then the constraint is satisfied. But if it is modified and the constraint is violated, then we move the outer circle so as to satisfy the constraint. The steps involved to modify the circle (inner or outer) are given below. The details of the derivation is given in Appendix H.

Let \( c_1 \) and \( c_2 \) be two circles with centres \( C_1 = (x_1', y_1') \) and \( C_2 = (x_2', y_2') \) respectively. Further, let us assume that \( c_1 \) contains \( c_2 \).

**Case 1:** Suppose that the circle \( c_1 \) is modified to \( c'_1 \) with centre \( C'_1 = (x'_1, y'_1) \) and radius \( r'_1 \) and the constraint is not satisfied. Then we move \( c_2 \) to \( c'_2 \) with centre \( C'_2 = (x'_2, y'_2) \) and radius \( r'_2 \) satisfying the constraint. The steps followed to determine \( C'_2 \) and \( r'_2 \) are given below:

**Step 1:** Test whether \( c'_1 \) contains \( c'_2 \). If yes, then the constraint is satisfied else go to step 2.

**Step 2:** Define the line \( L' \) passing through \( C'_1 \) and parallel to \( C_1C_2 \).

**Step 3:** Define \( r'_2 \) such that
\[
\frac{r'_2}{r'_1} = \frac{r_2}{r_1}
\text{ holds.}
\]

Step 4: Define the point \( C'_2 \) lying on \( L' \) and in the direction of \( C_1C_2 \) such that

\[
\frac{d(C_1, C_2)}{r_1} = \frac{d(C'_1, C'_2)}{r'_1}
\]

is true.

i.e. define \( C'_2(x'_2, y'_2) \) and radius \( r'_2 \) as follows:

\[
x'_2 = x'_1 + \left( \frac{r'_1}{r_1} \right) (x_2 - x_1),
\]

\[
y' = y'_1 + \left( \frac{r'_1}{r_1} \right) (y_2 - y_1)
\]

and

\[
r'_2 = \left( \frac{r'_1}{r_1} \right) r_2
\]

Case 2: Assume that the circle \( c_2 \) is modified to \( c'_2 \) and the constraint is not satisfied, then we move \( c_1 \) to \( c'_1 \) so as to satisfy the constraint.

The following steps are considered to determine \( C'_1(x'_1, y'_1) \) and \( r'_1 \).

Step 1: If \( c'_2 \) still contains \( c_1 \) then the constraint is satisfied else go to step 2.

Step 2: Define \( r'_1 \) as:

\[
r'_1 = \left( \frac{r_1}{r_2} \right) r'_2
\]

Step 4: Define the point \( C'_1 \) lying on the line \( L' \) passing through \( C'_2(x'_2, y'_2) \) and parallel to \( C_2C_1 \) such that

\[
\frac{d(C'_1, C_2)}{r'_1} = \frac{d(C_1, C_2)}{r_1}
\]

is satisfied.
In fact, the centre $C'_1(x'_1, y'_1)$ of $C'_1$ is defined by:

$$x'_1 = x'_2 + \left( \frac{r'_1}{r_1} \right) (x_1 - x_2),$$

$$y'_1 = y'_2 + \left( \frac{r'_1}{r_1} \right) (y_1 - y_2).$$

### 4.2.3 Structural Constraints

We define structural constraints as the constraints that are imposed among components in a part hierarchy. Structural constraints are special case of inter-object constraints. As these constraints do not apply to all objects of a class, we use constraint objects as discussed in Section 4.2.2 to impose these constraints. For example, it is not correct to say that all wheels must refer to the same cylinder shape. We have further identified three types of constraints as follows.

#### 4.2.3.1 Geometric and Appearance Constraints

These are the constraints imposed on the geometry (position, orientation and size) and on the appearance (colour, line style) of an object on the screen. For example, the radius of the hub of a front wheel of a tricycle cannot be smaller than a given number.

get-radius(get-geometry(h)) $\geq$ 10, $h \in$ HUB

To impose these restrictions constraint objects are created as instances of a class as defined below:

**Class** Hub$\_$radius

$h$ : HUB

$\$radius : real
When the above constraint is imposed on a hub object, the radius of the hub object is stored in the database. The user has to create the hub object with the desired radius and then impose the constraint that the radius has to be less than or equal to the value stored in the database.

Another example is that the colour of the hub must always be red.

\[ \text{get-colour}(h) = '\text{red}', \ h \in \text{HUB} \]

To impose these constraints, constraint objects are created as instances of a class defined as below:

**Class Hub$colour**

\[
\begin{align*}
\text{h} &: \text{HUB} \\
\$\text{colour} &: \text{string}
\end{align*}
\]

With this constraint imposed on a hub object, the colour of the hub object is stored in the database. The user has to create the hub object with the 'red' colour and then impose the restriction that the colour of the object cannot change.

**4.2.3.2 Connectivity Constraints**

These constraints impose conditions on how two components are connected in the parent object. For example, the centre of base of the cylinder used to represent the rim of a wheel should coincide with that of the centre of the sphere representing hub.

\[ \text{get-centre(get-geometry}(w) = \text{get-centre(get-geometry}(h)), \]

\[ w \in \text{WHEEL}, \ h \in \text{HUB} \]
To impose these restrictions constraint objects are created as instances of class as defined below:

Class WH$centre
w : WHEEL
h : HUB

When this constraint is imposed on the wheel and hub objects, it is ensured that the centre of the wheel object is the same as that of hub. The user has to create the objects with the same centre and then impose the constraint that the centres of the two objects must always be equal.

Another example is that the radius of the front wheel must not be less than or at most be 1.5 times greater than that of hind wheels.

\[
\text{get-radius(get-geometry(w1))} \\
\leq \text{get-radius(get-geometry(w2))} \\
\leq 1.5 \times \text{get-radius(get-geometry(w1))}
\]

w1, w2 € WHEEL

To impose these restrictions constraint objects are created as instances of class as defined below:

Class WW$radius
whfront : WHEEL
whhind : WHEEL

4.2.3.3 Sharing Constraints

These constraints ensure that two or more objects share a geometric shape. For example, the component 'rim' of hind wheels of a tricycle must share a cylinder object, i.e., they must refer to the same cylinder object.
get-geometry(w1) = (get-geometry(w2)),
    w1, w2 € WHEEL

To impose these restrictions constraint objects are created as instances of class as defined below:

**Class WW$geometry**

whleft : WHEEL
wrighth : WHEEL

### 4.3 Classes and Constraints

For readability and easy referencing, the function names in the chapter have the following format.

```
ttt-n(-aaa)
```

where

- **ttt** indicates type of constraint
  - DOM : DOMain Constraint
  - IRA : IntRA-object constraint
  - SOB : Set Object Bound constraint
  - ASC : ASsoCiation constraint
  - IER : IntER-object constraint
  - REF : REFerence Constraint
  - INS : INStance Constraint
  - SGA : Structural Geometric and Appearance Constraint
  - SCN : Structural CoNnectivity Constraint
  - SSH : Structural SHaring Constraint

- **n** indicates that the constraint is invoked at the time of Insertion, Modification or Deletion.
  - I : Insertion
  - M : Modification
  - D : Deletion
Class WINDOW
constraints
DOM-I(w:WINDOW) → boolean
DOM-M-xsize(w:WINDOW) → boolean
DOM-M-ysize(w:WINDOW) → boolean
SOB-I(w:WINDOW) → boolean
SOB-M-destination(w:WINDOW) → boolean
ASC-I(w:WINDOW) → boolean
ASC-M-destination(w:WINDOW) → boolean
REF-D(w:WINDOW) → boolean

Class VIEWPORT
constraints
DOM-I(v:VIEWPORT) → boolean
DOM-M-xsize(v:VIEWPORT) → boolean
DOM-M-ysize(v:VIEWPORT) → boolean
SOB-I(v:VIEWPORT) → boolean
SOB-M-origin(v:VIEWPORT) → boolean
ASC-I(v:VIEWPORT) → boolean
ASC-M-origin(v:VIEWPORT) → boolean
REF-D(v:VIEWPORT) → boolean
IER-I(v:VIEWPORT) → boolean
IER-M-xsize(v:VIEWPORT) → boolean
IER-M-ysize(v:VIEWPORT) → boolean

Class DEVICE
constraints
DOM-I(d:DEVICE) → boolean
DOM-M-xmaxsize(d:DEVICE) → boolean
DOM-M-ymaxsize(d:DEVICE) → boolean
REF-D(d:DEVICE) → boolean

Class POINT
constraints
INS-M-pp$distance(p1,p2:POINT) → boolean
INS-D-pp$distance(p1,p2:POINT) → boolean

Class LINE
constraints
IRA-I(l:LINE) → boolean
IRA-M-tail(l:LINE) → boolean
IRA-M-head(l:LINE) → boolean

Class TEXT
constraints
DOM-I(t:TEXT) → boolean
DOM-M-height(t:TEXT) → boolean

Class CIRCLE
constraints
DOM-I(c:CIRCLE) → boolean
DOM-M-radius(c:CIRCLE) → boolean
A database system must ensure that the data is valid with respect to a set of constraints by enforcing the constraints whenever the database is updated. The enforcement activity is called constraint validation.

The constraints are validated whenever an update operation i.e. (Insert, Modify, Delete) takes place. We use functions to maintain constraints which are automatically invoked and executed at the time of insertion, deletion and modification. These functions are not called directly by the user but are invoked by the respective update operations.

The constraints can also be validated at the time of completion of transaction, at the time of completion of particular design phase, at a particular time, or as mentioned above at the time of insertion, deletion and modification. In this chapter we focus on the last case.
Maintenance is a set of actions to be performed in case of constraint violation and/or satisfaction. Functions detecting a constraint violation returns a boolean value depending on whether the constraint is satisfied or not and accordingly different actions may be taken [Borning 1979], [Laufe 1986], [Qian 1986].

In [Lyngbaek 1989], modification is modelled by deletion followed by an insertion. Treating modifications as above is an overhead because inserting an object may require several constraints to be validated. Further deletion of an object may be aborted because of deletion constraint imposed on it. Also, the deletion of an object may result in a trigger delete. Therefore, we treat modification separately.

4.4.1 Maintenance of Class Constraints

4.4.1.1 Domain Constraints

For all instance variables on which domain constraints are imposed, a function is invoked to validate the constraint at the time of insertion. The object to be inserted is passed as an argument to the function. For modification, a function exists for each instance variable on which domain constraint is imposed. The modified object which is in memory is passed as an argument to this function. If the constraint is satisfied the modifications are stored in the database. The constraint validation functions for insertion do not access the
database.

function DOM-I(r:RECTANGLE) → boolean;
{ This constraint ensures that the xlength and ylength } 
{ of a rectangle object is greater than 0. } 
{ This constraint is invoked at the time of insertion. } 
if ( (get-xlength(r) > 0) AND (get-ylength(r) > 0) ) 
then DOM-I ← true 
else DOM-I ← false

function DOM-M-xlength(r:RECTANGLE) → boolean; 
{ This constraint ensures that the xlength } 
{ of a rectangle object is greater than 0. } 
{ This constraint is invoked at the time of modification } 
if ( get-xlength(r) > 0) 
then DOM-M-xlength ← true 
else DOM-M-xlength ← false

4.4.1.2 Intra-object Constraints

For each set of instance variables on which intra-object constraints are specified, there are functions for insertion and modification. For validating the constraint at the time of insertion, the object to be inserted is passed as an argument to this function whereas in case of modification the modified object is passed as an argument. The validation functions for insertion do not access the database.

function IRA-I(l:LINE) → boolean;
{ This constraint ensures that the line object l has } 
{ distinct end points. } 
{ This constraint is invoked at the time of insertion. } 
p1 ← get-tail(l); 
{ p1 is a POINT } 
p2 ← get-head(l); 
{ p2 is a POINT } 
temp ← equal(p1,p2); 
{ temp is true if points p1 and p2 have same } 
{ x and y coordinates. } 
if (NOT temp)
then IRA-I \leftarrow true
\text{.}
else IRA-I \leftarrow false

\text{function IRA-M-tail(I:LINE) \rightarrow boolean ;}
\text{(This constraint ensures that when tail of the line )}
\text{(object I is modified, it does not coincide with )}
\text{(head of the same line. )}
\text{(This constraint is invoked at the time of modification.)}
\text{(The code is same as that of IRA-I )}

\textbf{4.4.1.3 Set Object Bound Constraints}

If a class has set object bound constraint defined on
an instance variable, a function is invoked and executed
to validate the constraint for insertion. The object to
be inserted is passed as an argument. For modification,
the modified object is passed as an argument. The
validation functions for insertion do not access the
database.

\text{function SOB-I(w:WINDOW) \rightarrow boolean ;}
\text{(This constraint ensures that the number of )}
\text{(destinations of the window object lies between )}
\text{(0 and 10 both inclusive. )}
\text{(This constraint is invoked at the time of insertion. )}
\text{VP \leftarrow destination(w) ;}
\text{(VP is the set of destinations of object w. )}
\text{n \leftarrow count(VP) ;}
\text{(n is the number of members in set VP. )}
\text{if ( (n \geq 0) AND (n \leq 10) )}
\text{then SOB-I \leftarrow true}
\text{else SOB-I \leftarrow false}

\text{function SOB-M-destination(w:WINDOW) \rightarrow boolean ;}
\text{(This constraint ensures that the number of )}
\text{(destinations of the window object lies between )}
\text{(0 and 10 both inclusive. )}
\text{(This constraint is invoked at the time of modification.)}
\text{(The code is same as that of SOB-I )}
4.4.1.4 Association Constraints

For each pair of instance variables on which association constraint is specified, there is a function to maintain the constraint at the time of insertion. The object to be inserted is passed as an argument to this function which automatically modifies the instances of the associated class. Similarly, to maintain the constraint at the time of modification, the object and the modified object are passed as arguments to this function. This function also automatically modifies the instances of the associated class. The constraint maintenance functions have to access the database. Association constraints are two-way constraints.

function ASC-I(w:WINDOW) \rightarrow boolean ;
{ This constraint ensures that the window object w }
{ is added to the 'source' of the viewport objects }
{ on which w is mapped. }
{ This constraint is invoked at the time of insertion. }
VP \leftarrow get-destination(w) ;
{ VP is the set of viewport objects on which w is mapped }
i \leftarrow 1 ;
success \leftarrow true ;
while ( (NOT(empty(VP))) AND (success) )
begin
  v \leftarrow element(i,VP) ;
  { v is a viewport object }
  remove(v,VP) ;
  WN \leftarrow origin(v) ;
  { WN is the 'origin' of v and is the set of window }
  insert(w,WN) ;
  success \leftarrow modify(v,'origin',WN) ;
end
if (success)
then ASC-I \leftarrow true ;
else ASC-I \leftarrow false ;
function ASC-M-destination(w,wupd:WINDOW) \rightarrow boolean;
{ This constraint ensures that the window object w }
{ is added to the 'source' of the viewport objects }
{ on which w is mapped. }
{ This constraint is invoked at the time of modification.}
VOLD \rightarrow destination(w);
{ VOLD is the set of viewports in the old object }
VNEW \rightarrow destination(wupd);
{ VNEW is the set of viewports in the new object }
V1 \rightarrow difference(VOLD,VNEW);
V2 \rightarrow difference(VNEW,VOLD);
if ( empty(V1) AND empty(V2) )
then
{ This path implies VOLD and VNEW have same members }
{ and hence the constraint is satisfied. }
ASC-M-destination \rightarrow true
else begin
if (not(empty(V1)))
then begin
{ In this block viewport objects belonging to V1 are }
{ removed from the set of viewports of the }
{ corresponding window objects. }
success \rightarrow true;
i \rightarrow 1;
while ( (not(empty(V1))) AND (success) )
begin
v \rightarrow element(i,V1);
remove(v,V1);
WN \rightarrow origin(v);
remove(w,WN);
success \rightarrow modify(v,'origin',WN);
end
end
if ( (not(empty(V2))) AND (success) )
then begin
{ In this block viewport objects belonging to V2 }
{ are inserted in set of viewports of the }
{ corresponding window objects. }
i \rightarrow 1;
while ( (not(empty(V2))) AND (success) )
begin
v \rightarrow element(i,V2);
remove(v,V2);
WN \rightarrow origin(v);
insert(w,WN);
success \rightarrow modify(v,'origin',WN);
end
end
if (success)
then ASC-M-destination \rightarrow true
else ASC-M-destination \rightarrow false
end
4.4.1.5 Inter-Object Constraints

If a class has an inter-object constraint specified, a function is invoked and executed at the time of insertion and modification. The object to be inserted or modified is passed as an argument to this function. The validation functions have to access the database.

```
function IER-I(v:VIEWPORT) ➔ boolean ;
{ This constraint ensures that the xlength and }
{ ylength of the viewport lies within the }
{ xlength and ylength of the corresponding device.}
{ This constraint is invoked at the time of insertion. }
d ← get-device(v) ;
loc ← get-location(v) ;
{ loc is a POINT. }
xloc ← get-xcor(loc) ;
yloc ← get-ycor(loc) ;
xl ← get-xlength(v) ;
yl ← get-ylength(v) ;
xmd ← get-xmaxsize(d) ;
ymd ← get-ymaxsize(d) ;
xtemp ← xloc + xl ;
ytemp ← yloc + yl ;
if { (0 < xtemp) AND (xtemp ≤ xmd) AND
     (0 < ytemp) AND (ytemp ≤ ymd) )
then IER-I ← true
else IER-I ← false
```

```
function IER-M-xlength(v:VIEWPORT) ➔ boolean ;
{ This constraint ensures that the xlength and }
{ ylength of the viewport lies within the }
{ xlength and ylength of the corresponding device.}
{ This constraint is invoked at the time of modification.}
{ This function is same as IER-I except that it varifies }
{ xlength only. }
```

To satisfy inter-object and association constraints information about the internals of other objects are required. Although in any object-oriented programming, the instance variables of an object are hidden and therefore can not be accessed from other object’s methods
but we have provided functions that can access the instance variables of an object, and they can be invoked by functions defined on other objects.

4.4.1.6 Referential Integrity Constraints

For any constraint having "\( \exists \)" quantifier on a class, a function is invoked at the time of deletion of an instance of that class. The object to be deleted is passed as an argument to the function. The constraint maintenance functions have to access the database. The following functions are examples of the three approaches to enforce the constraints as discussed in section 4.2.1.6.

function REF-D(w:WINDOW) -> boolean;
{ This constraint ensures that the deletion of a window object is aborted if this is the only window mapped on a viewport. }
{ This constraint is invoked at the time of deletion. }
found <- false;
success <- true;
i <- 1;
VP <- get-destination(w);
if (empty(VP))
then REF-D <- true
else begin
  while ((NOT(empty(VP))) AND (NOT(found))) AND (success))
  begin
    v <- element(i,VP);
    remove(v,VP);
    WN <- get-origin(v);
    n <- count(WN);
    if (n>1)
      then begin
        insert(w,WN);
        success <- modify(v,'origin',WN);
        if (success)
          then REF-D <- true
          else REF-D <- false
      end
  end
end
else begin
    found ← true;
{ A viewport object has only w mapped on it. }
    REF-D ← false;
{ The constraint returns false. }
end

function REF-D (d:DEVICE) → boolean;
{ This constraint ensures that when a device }
{ is deleted, all objects that refer to it }  
{ are also deleted. }
for each v in VIEWPORT
begin
    dtemp ← get-device(v);
    if (same(d,dtemp))
    then delete(y)
end
    REF-D ← true;

function REF-D (v:VIEWPORT) → boolean;
{ This constraint ensures that when a viewport is } 
{ deleted, all references to it are set to null. }
for each w in WINDOW
begin
    VP ← get-destination(w);
    if (member(v,VP))
    then begin
        remove(v,VP);
        success ← modify(w,’destination’,VP);
    end
end

4.4.2 Maintenance of Instance Constraints

A function exists corresponding to each class on which instance constraint is specified. The constraint has to be checked at the time of modification and deletion. At the time of modification, a function is invoked which first determines the existence of an instance constraint on the object and then verifies whether the modified object satisfies the constraint. If
the constraint is not satisfied then the other object is modified in order to satisfy the constraint. In case of deletion, if there is an instance constraint specified on an object, then the deletion of the object is aborted. The validation functions have to access the database. As mentioned before, instance constraints are two-way constraints.

function INS-M-PP$distance(p,pupd:POINT) → boolean;
{ This function first determines the existence of an }
{ instance constraint on point object p. }
{ If there is an instance constraint imposed on the }
{ object and the modified object does not satisfy the }
{ constraint, then we move the other object so as to }
{ satisfy the constraint. }
{ This constraint is invoked at the time of modification. }
for each cp in PP$distance
{ cp is a constraint object belonging to class }
{ PP$distance. }
    p1 ← point1(cp);
{ p1 is a POINT }
    p2 ← point2(cp);
{ p2 is a POINT }
    d ← distance(cp);
    temp ← same(p,p1);
    if (temp)
        then begin
            dnew ← distance(pupd,p2);
            if (d=dnew)
                then IC-M-PP$distance ← true
                else begin
                    move(p2)
                    end
            end
    else begin
        temp ← same(p,p2)
        if (temp)
            then begin
                dnew ← distance(pupd,p1);
                if (d=dnew)
                    then INS-M-PP$distance ← true
                    else begin
                        move(pl)
                    end
            end
end
{ Point p2 is modified to satisfy the constraint }
{ using algorithm given in section 4.2.2.1}
INS-M-PP$distance ← true
end
{ Point pl is modified to satisfy the constraint }
{ using algorithm given in section 4.2.2.1}
INS-M-PP$distance → true
end
end

function INS-D-PP$distance(p : POINT) → boolean;
{ This function first determines the existence of an }
{ instance constraint on point object p. }
{ If there is an instance constraint imposed on the }
{ object, then the function returns false, i.e., the }
{ deletion of the object is aborted. }
{ This constraint is invoked at the time of deletion. }
for each cp in PP$distance
{ cp is a constraint object belonging to class }
{ PP$distance. }
begin
p1 ← point1(cp);
{ p1 is a POINT }
p2 ← point2(cp);
{ p2 is a POINT }
temp1 ← same(p,p1);
temp2 ← same(p,p2);
if (temp1 OR temp2)
then INS-M-PP$distance ← false
else INS-M-PP$distance ← true
end

4.4.3 Maintenance of Structural Constraints

A function exists corresponding to each class on
which structural constraint is specified. The constraint
has to be checked at the time of modification and
deletion. At the time of modification, a function is
invoked which first determines the existence of a
structural constraint on the object and then verifies
whether the modified object satisfies the constraint. In
case of deletion, if there is an structural constraint
specified on an object, then the deletion of the object is
aborted. The validation functions have to access the
database. We give examples using sharing constraints.
Constraint maintenance functions for other types of
structural constraints can be derived in the similar way.

function SSH-M-WW$geometry(w, wupd: WHEEL) \rightarrow boolean;
{ This function first determines the existence of an 
instance constraint on wheel object w. }
{ The function then verifies that the left and right }
{ wheels refer to the same object. }
{ This constraint is invoked at the time of modification. }
for each cp in WW$geometry
{ cp is a constraint object belonging to class }
{ WW$geometry. }
  w1 \leftarrow \text{whleft}(cp);
{ w1 is a WHEEL object }
  w2 \leftarrow \text{whright}(cp);
{ w2 is a WHEEL object }
  temp \leftarrow \text{same}(w, w1);
if (temp)
then begin
{ Constraint is imposed on wheel object w. }
  gl \leftarrow \text{geometry}(w1);
  gnew \leftarrow \text{geometry}(w);
if (gl=gnew)
then SSH-M-WW$geometry \leftarrow true
else SSH-M-WW$geometry \leftarrow false
end
else begin
  temp \leftarrow \text{same}(w, w2)
if (temp)
then begin
{ Constraint is imposed on wheel object w. }
  g2 \leftarrow \text{geometry}(w2);
  gnew \leftarrow \text{geometry}(w);
if (g2=gnew)
then SSH-M-WW$geometry \leftarrow true
else SSH-M-WW$geometry \leftarrow false
end
end

function SSH-D-WW$geometry(w : WHEEL) \rightarrow boolean;
{ This function first determines the existence of an }
{ instance constraint on point object p. }
{ If there is an instance constraint imposed on the }
{ object, then the function returns false, i.e., the }
{ deletion of the object is aborted. }
{ This constraint is invoked at the time of deletion. }
for each cp in WW$geometry
{ cp is a constraint object belonging to class }
{ WW$geometry. }
begin
  w1 \leftarrow \text{whleft}(cp);
{ w1 is a WHEEL object }
  w2 \leftarrow \text{whright}(cp);
{ w2 is a WHEEL object }
\[\text{templ} \rightarrow \text{same}(w,w1)\]
\[\text{temp2} \rightarrow \text{same}(w,w2)\]
\[\text{if (temp1 OR temp2)}\]
\[\text{then SSH-M-WW$\!\!\!\!\!\!\!$geometry} \rightarrow \text{false}\]
\[\text{else SSH-M-WW$\!\!\!\!\!\!\!$geometry} \rightarrow \text{true}\]

4.5 Operations and Constraints

As discussed in sections 4.3 and 4.4, insert, delete and modify operations invoke several constraints. This section summarize the constraints invoked by these operations and the action taken for different classes of objects.

<table>
<thead>
<tr>
<th>Operation</th>
<th>INSERT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object type</td>
<td>Constraint type</td>
</tr>
<tr>
<td>Window</td>
<td>DOM, SOB</td>
</tr>
<tr>
<td></td>
<td>ASC</td>
</tr>
<tr>
<td>Viewport</td>
<td>DOM, SOB</td>
</tr>
<tr>
<td></td>
<td>ASC</td>
</tr>
<tr>
<td>Device</td>
<td>DOM</td>
</tr>
<tr>
<td>Primitive</td>
<td>DOM, IRA</td>
</tr>
<tr>
<td>Object type</td>
<td>Constraint type</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Window</td>
<td>REF</td>
</tr>
<tr>
<td>Viewport</td>
<td>REF</td>
</tr>
<tr>
<td>Device</td>
<td>REF</td>
</tr>
<tr>
<td>Primitive</td>
<td>INS</td>
</tr>
<tr>
<td>Complex</td>
<td>SGA, SCN, SSH</td>
</tr>
<tr>
<td>Object type</td>
<td>Constraint type</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Window</td>
<td>DOM, SOB</td>
</tr>
<tr>
<td></td>
<td>ASC</td>
</tr>
<tr>
<td>Viewport</td>
<td>DOM, SOB</td>
</tr>
<tr>
<td></td>
<td>ASC</td>
</tr>
<tr>
<td>Primitive</td>
<td>DOM, IRA</td>
</tr>
<tr>
<td></td>
<td>INS</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>SGA, SCN, SSH</td>
</tr>
</tbody>
</table>