# Constraint solver for generalized IC layout

by Peter W. Cook

**This paper presents a constraint solver suitable for use in a general symbolic IC layout system. The essential features of the constraint solver, which is intended to place few restrictions on the source of the constraints to be solved, are that it accommodate mixed equality and inequality constraints, that it allow selective "maximization" of variables, that it proceed with any number of variables given user-defined values, and that it fail to produce a solution only when no solution exists. These features all flow from the desire to provide a constraint solver suitable for use in an "open" system, in which there are no restrictions on the form or order of the constraints. The algorithm presented meets these objectives while remaining reasonable in its use of storage and time. An extension to the class of constraints acceptable by the constraint solver is presented; the extension of the system to this added constraint class has yet to be done.**

## Introduction

Within recent years, a number of design-rule-"independent" layout systems have been developed to facilitate mask designs for integrated circuits [1–5]. Such systems do not achieve total independence from design rules, but do achieve significant independence from design rule values. One way that this can be done is illustrated in **Figure 1**. Figure 1(a) shows a "typical" polysilicon gate structure, and a symbolic "shapes" description: clearly, if values for the various coordinates (X1, X2, . . .) and mask levels (ROX, POLY, . . .) can be determined, the part can be constructed from the "shapes" description, which remains valid for any (design-rule-legal) coordinate values. Figure 1(b) shows one method for expressing the design rules. This figure reflects the fact that most rules specify a lower bound on the distance between two edges in the pattern to be created. In Fig. 1(b), a "design.rules" block provides a list of the relationships imposed by a given kind of technology; since the values associated with any particular constraint in this set of rules are given symbolically, this description also remains valid as the design rule values are changed. Thus, the description formed by the "shapes" and "design.rules" blocks is a general description of the polysilicon gate. An embodiment of the design rules in a specific technology appears in Fig. 1(c); here, the technology's design rule values have been substituted for the symbolic design rules of Fig. 1(b). While this example is simple, it is clear that the concept generalizes to complex shapes. A system implementing such a mechanism for shapes description will allow shapes to be generated for relatively arbitrary instances of a class of design rules; specifically, it will allow valid parts to be generated in the most recent version of such design rules without the need for manual intervention. Where such a system includes expressions [such as the last rule relating X6 and X7 in Fig. 1(b)], parts can be parameterized to allow the designer simple control over device characteristics.

What is needed to make such a system work is a constraint solver capable of handling the relationships in the "design.rules" block; while most design rules result in inequality relationships, some rules result in exact equalities: this is essential in MOSFET technologies, where the width

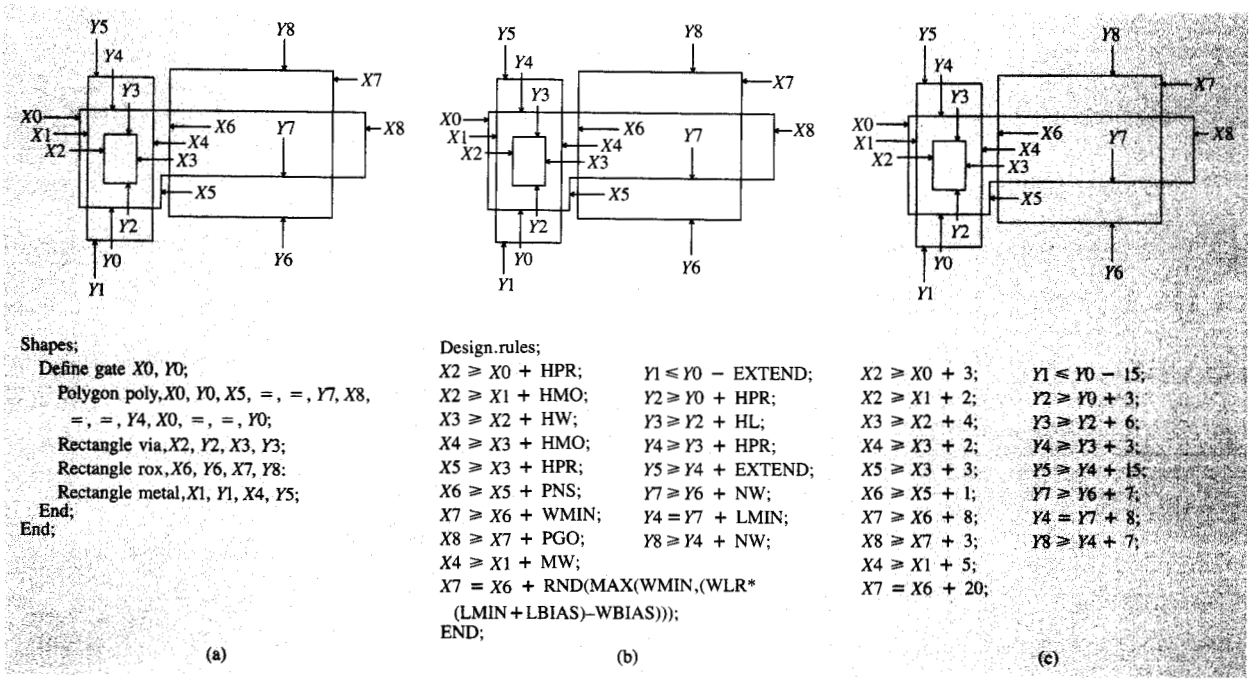


**Figure 1**

(a) General definition of a set of shapes. (b) Constraints for previous shapes. (c) Constraints from previous example in specific technology.

and length of device "channels" determine the transconductance of the device, but can occur in any technology. Thus the system must, in solving the constraint problem, accommodate exact equalities as well as the more typical inequalities.

A second significant feature of the constraint solver can be seen by examining the diagrams in **Figure 2**. It is fairly typical to program a constraint solver to set all solved values to their lower bound. In Fig. 2(a) another version of the polysilicon gate is shown; Fig. 2(b) displays the result of using all minimum values in this part: the wire contacting the gate has become excessively wide. This can be corrected by giving the left edge of the wire the maximum legal value it can have [Fig. 2(a)]. "Maximum" here should be taken as having a strictly limited context: the "maximized" variable is made as large as possible without moving already "minimized" edges.

This paper describes in detail the constraint solver that is in use in a symbolic layout system at the IBM T. J. Watson Research Center. The constraint solver includes the features outlined above, and operates with reasonable time and space requirements.

## Basic approach

Generally, inequality constraints may be combined uniformly into either ≥ or ≤ constraints by using the fact that the constraints

$$X_a \le X_b + C_{ab},$$

$$X_b \ge X_a - C_{ab},$$

are equivalent. Thus, all of the inequality constraints in a problem may be expressed as one type.

Equality constraints can be expressed as a pair of inequalities: thus,

$$X_b = X_a + C_{ab}$$

is identical to the pair

$$X_b \ge X_a + C_{ab},$$

$$X_b \le X_a + C_{ab}.$$

This approach can be taken in the constraint solver [6]; however, it leads to an increase in the total number of constraints, and may "bury" an inconsistent equality constraint in the inequality constraints as a cyclic problem, making location of the error somewhat more difficult. The present approach prevents this by solving the equality constraints before any inequalities are examined. It is also tolerant of any constraint set that can be solved.

The basic concept in the present algorithm is to make extensive use of relaxation techniques, since they allow efficient storage of the problem to be solved. Equalities are handled by treating them as the definition of one unknown in terms of another; the definition is substituted in all

inequality constraints, while retaining the equality relationship. This leads to a smaller inequality problem, and also allows early detection of inconsistent equality constraints.

The constraint solver briefly can be thought of as operating in several steps. First, the equality constraints are grouped together, and each equality subgraph is solved in terms of an arbitrarily selected node of the subgraph. Second, these solutions are substituted back into the remaining inequality relationships. Third, these modified inequalities are solved by a relatively simple relaxation technique. Fourth, the relative solutions of the equalities are used to set values of variables earlier substituted out of the inequality relationships.

## Notation

In discussing the constraint solver in detail, we make use of the fact that systems of constraints have a natural representation in directed graphs. Each "vertex" of the graph corresponds to a variable in the constraint problem; each edge of the graph corresponds to a single constraint. The graph edge corresponding to a given constraint is placed between the graph vertices corresponding to the variables in the constraint; the direction of the edge for a constraint of the form

$$X_b \geq X_a + C_{ab}$$

or

$$X_b \leq X_a + C_{ab}$$

is directed "from" the vertex corresponding to $X_a$ "to" the vertex corresponding to $X_b$. Associated with each edge is a weight (the value of the constraint) and a type (the type of the constraint).

Within this description, the notation is kept close to that actually used in the program. Constraints and the resulting solution are stored in two tables: a vertex table and an edge table. Besides symbolic name information (which is not relevant to the algorithm), the vertex table contains the following:

$X(v)$: the value associated with vertex $v$.
$D(v)$: a boolean signifying that the value is an external definition.
$RV(v)$: the "relative" value of vertex $v$, when involved in equalities.
$R(v)$: the "representative" of vertex $v$.
$AL(v)$: the "assignment level" associated with vertex $v$.

The edge table describes constraints through pointers to elements in the vertex table. The edge table contains the following:

$T(e)$: a pointer to the "to" vertex of edge $e$.

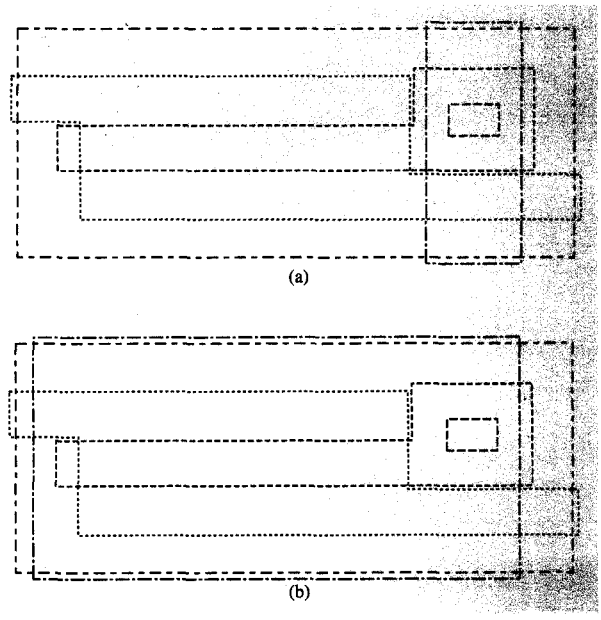


**Figure 2**

(a) Expected results. (b) "All minimum" results.

$CT(e)$: the type of constraint represented by edge $e$: $\geq$, $\leq$, $=$, or *NULL*.
$F(e)$: a pointer to the "from" vertex of edge $e$.
$CV(e)$: the value associated with edge $e$.

Thus edge $e$ signifies that

$$X[T(e)]\ CT(e)\ X[F(e)] + CV(e),$$

where $CT(e)$ signifies $\geq$, $\leq$, or $=$. (The *NULL* constraint type has no interpretation as a relationship, and is used only temporarily, to signify that the constraint represented by an edge has been incorporated in the solution, and hence the edge is no longer needed.) Note that the storage requirements are linear in the number of vertices and edges.

## Algorithm details

The details of the algorithm are now described using the notation introduced earlier, and by means of an example.

- *Phase 1: Process equalities*

The first major task of the constraint solver is to locate and "solve" all equality constraints. Consider the original constraint graph ($G_1$, **Figure 3**). A second graph ($G_2$) can be formed from $G_1$ by (conceptually) deleting all inequality constraints. **Figure 4** illustrates the result; in the figure, $G_2$ contains one subgraph (connecting vertices X6 and X7); generally, several disjoint subgraphs are found in $G_2$. Each

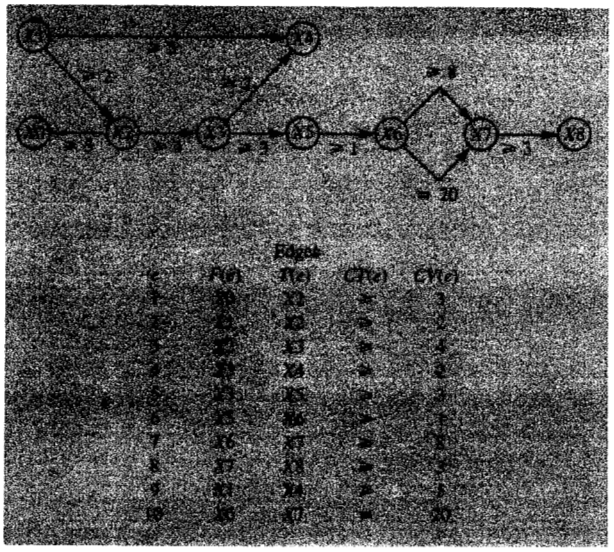

**Figure 3**

Sample graph ($G_1$) and its tabular representation. This is the $X$ graph from the part of Figure 1.

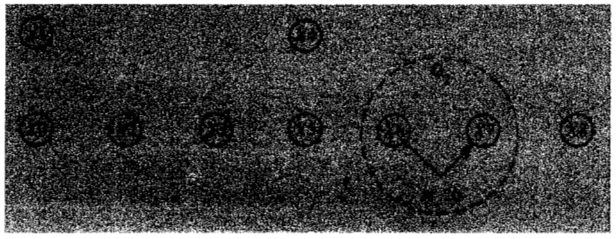

**Figure 4**

Equality subgraph for graph of Figure 3. Only vertices $X6$ and $X7$ are on the subgraph.

disjoint subgraph of $G_2$ is termed an equality subgraph. Assuming that the constraints in an equality subgraph are consistent, any vertex on such a subgraph may be expressed in terms of any other vertex on the same subgraph by

$$X(a) = X(b) + C_{ab}.$$

More specifically, one vertex on each equality subgraph may be taken as "representative" of that subgraph; all other vertices on the subgraph may be expressed in terms of the representative vertex by

$$X(v) = X(r) + RV(v),$$

where $r$ is a pointer to the representative vertex, $v$ is a pointer to a vertex on the same equality subgraph as $r$, and $RV$ gives the value of $v$ relative to $r$.

This phase of the constraint solver locates all equality subgraphs, selects a representative vertex for each equality subgraph, and determines the relative values of all nonrepresentative vertices on each equality subgraph. It also reports failure when a set of equality constraints cannot be satisfied. The operation is as follows.

*Step 1. Begin representative search.*
A pointer (*RULE*) is set to zero.

*Step 2. Test for "=" constraint.*
Here, the solver looks for the next equality constraint. To do this, it increments the pointer (*RULE*) by one. After *RULE* has been incremented, one of three cases must apply:

a. *RULE* points beyond the last element in the edge table.
   In this case all equality subgraphs have been solved (relative to their own "representative" vertex), and this phase is completed.
b. *CT*(*RULE*) is not "=".
   This edge is not relevant. It is skipped, and the solver returns to Step 2 above.
c. *CT*(*RULE*) is "=".
   The constraint represented by the edge at *RULE* is an equality relationship. The solver proceeds to Step 3.

*Step 3. Set representative vertex.*
The vertex *F*(*RULE*) is arbitrarily identified as the representative vertex for this equality subgraph. (Since the vertices of the equality subgraph are all linked by equality relationships, this "arbitrary" representation involves no loss of generality.) To do this, the following assignments are made:

$$REP = F(RULE),$$
$$R(REP) = 0,$$
$$RV(REP) = 0,$$
$$R[T(RULE)] = REP,$$
$$RV[T(RULE)] = CV(RULE).$$

This completely expresses the equality constraint represented by the edge at *RULE*, so that *CT*(*RULE*) is set to *NULL*, effectively freeing the location in the edge table for later deletion. The constraint solver now locates all vertices that are linked to vertex *REP* by edges representing equality constraints. This is the equality subgraph associated with the vertex *REP*. At the same time, the constraint solver determines values for these vertices (in terms of the vertex *REP*) resulting from the equality constraints. This is done by a simple relaxation algorithm that examines only the edges representing equality constraints.

*Step 4. Begin pass on this subgraph.*
A counter variable (*COUNT*) is set to 0. A second edge pointer (*NEXT*) is set to *RULE*.

*Step 5. Scan for next "=" constraint.*
*NEXT* is incremented by 1. After this, as in Step 1 above, one of three situations is encountered:

a. *NEXT* points past the end of the edge table.
   All edges have been examined; the constraint solver enters Step 7 below.
b. *CT*(*NEXT*) is not "=".
   The constraint represented by the edge at *NEXT* is not relevant; the constraint solver returns to Step 5 above to examine another edge.
c. *CT*(*NEXT*) is "=".
   The constraint represented by edge *NEXT* may be part of the same equality subgraph containing *RULE*, or it may be a part of another equality subgraph. The solver enters Step 6 to determine this.

*Step 6. Implement related "=" constraint.*
The constraint at *NEXT* is of type "=". It may or may not be related to the subgraph currently in process, depending upon whether or not either vertex of *NEXT* is *REP* or represented by *REP*. When the vertices *T*(*NEXT*) and *F*(*NEXT*) are examined, one of four conditions arises:

a. Neither *T*(*NEXT*) nor *F*(*NEXT*) are *REP* or are represented by *REP*.
   In this case, nothing has yet been encountered to link a vertex of this edge to the equality subgraph being processed. This edge is skipped, and control passes to Step 5.
b. *F*(*NEXT*) is *REP* or represented by *REP*, *T*(*NEXT*) is neither.
   In this case, this edge is part of the equality subgraph since vertex *F*(*NEXT*) is already included in the subgraph. *T*(*NEXT*) therefore becomes a part of the subgraph, and will be defined in terms of *REP*. Thus, the following assignments are made:

   $$R[T(NEXT)] = REP,$$
   $$RV[T(NEXT)] = RV[F(NEXT)] + CV(NEXT),$$
   $$COUNT = COUNT + 1,$$
   $$CV(NEXT) = NULL,$$

   where the last step is justified because the entire content of the constraint represented by the edge *NEXT* has been embedded in *R*[*T*(*NEXT*)] and *RV*[*T*(*NEXT*)]. Control is passed to Step 5, to examine the next edge.
c. *T*(*NEXT*) is *REP* or represented by *REP*, *F*(*NEXT*) is neither.
   This is the reverse of the situation in (b); in this case, *F*(*NEXT*) joins the subgraph because *T*(*NEXT*) is already a member of the subgraph. The assignments

   $$R[F(NEXT)] = REP,$$
   $$RV[F(NEXT)] = RV[T(NEXT)] - CV(NEXT),$$
   $$COUNT = COUNT + 1,$$
   $$CV(NEXT) = NULL$$

   implement this, and control passes to Step 5.
d. *T*(*NEXT*) and *F*(*NEXT*) are both *REP* or represented by *REP*.
   In this case, both vertices *T*(*NEXT*) and *F*(*NEXT*) are already members of the equality subgraph; both have been expressed in terms of the vertex *REP*. This prior solution gives $\{RV[T(NEXT)] - RV[T(NEXT)]\}$ as the difference in values of the vertices *T*(*NEXT*) and *F*(*NEXT*). Two conditions exist:
   1. $\{RV[T(NEXT)] - RV[F(NEXT)]\} = CV(NEXT)$.
      The constraint represented by the edge at *NEXT* is redundant. Therefore *CT*(*NEXT*) is set to *NULL* for subsequent deletion, and control passes to Step 5.
   2. $\{RV[T(NEXT)] - RV[F(NEXT)]\} \neq CV(NEXT)$.
      The constraint represented by the edge at *NEXT* is inconsistent with other equality constraints on this subgraph. The problem has no solution. The constraint solver, after a suitable message, terminates.

*Step 7. Check for modifications.*
When *NEXT* has reached the bottom of the edge table, this step checks for any modifications. There are two cases:

a. $COUNT > 0$.
   At least one vertex has been added to the equality subgraph associated with vertex *REP*. Because this added vertex may also be used in equality constraints, it is necessary to scan the edge list from *RULE* + 1 once again. Therefore, the constraint solver passes to Step 4 to begin another scan.
b. $COUNT = 0$.
   The equality subgraph associated with vertex *REP* is now complete. However, there may be other equality subgraphs, and these could include a constraint represented by any edge after *RULE*. Therefore the constraint solver returns to Step 2.

Phase 1 is completed only when the pointer *RULE* has reached the end of the edge table. At that point, all equality subgraphs have been located, and each subgraph has been associated with one of its vertices (the "representative" vertex). In addition, all vertices of each subgraph have been solved in terms of the representative vertex. This is equivalent to solving each equality subgraph under the assumption that the representative vertex has value 0. In general, of course, the representative vertex will not be 0, but as the inequalities cause vertices of an equality subgraph to move, all vertices of a given subgraph must move by the

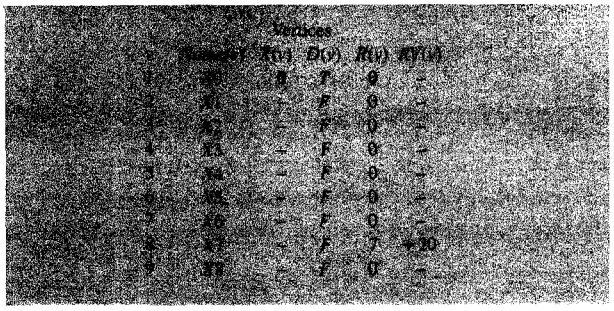

**Figure 5**

Partial vertex table after subgraph solution. $X7$ is now defined in terms of $X6$ ($X7 = X6 + 20$). The table also indicates that $X0$ has been defined to be zero.

same amount; thus $RV(v)$ gives the value of vertex $v$ relative to the value of vertex $R(v)$ [providing $R(v) > 0$]. The problem is now as illustrated in **Figure 5**.

• *Phase 2: Substitution*

Because all vertices on an equality subgraph are known in terms of the representative vertex associated with that subgraph, it is possible to substitute the representative vertex in all edges representing inequality constraints. The substitution phase accomplishes this. Suppose, for example, we have a given inequality ($e$),

$T(e)\ CT(e)\ F(e)\ CV(e)$,

and also suppose that as a result of Phase 1 we have

$R[F(e)] = j > 0$;

that is to say, $X[F(e)] = X(j) + RV[F(e)]$; in this case the constraint is modified by the assignments

$CV(e) = CV(e) + RV[F(e)]$,

$F(e) = R[F(e)]$.

The actual substitution takes place by scanning the edge table just once. For a typical edge ($e$), $R[F(e)]$ and $R[T(e)]$ are examined. Two cases are considered:

a. $R[F(e)] = 0$ and $R[T(e)] = 0$.
   Neither vertex $F(e)$ nor $T(e)$ is a part of any equality subgraph. There is no substitution to perform on this edge.
b. $R[F(e)] = j > 0$, or $R[T(e)] = k > 0$.
   At least one vertex of edge $e$ is a member of an equality subgraph. The following assignments are therefore made:
   1. If $R[F(e) = j > 0$, set
      $CV(e) = CV(e) + RV[F(e)]$,
      $F(e) = j$.
   2. If $R[T(e)] = k > 0$, set
      $CV(e) = CV(e) - RV[T(e)]$,
      $T(e) = k$.

An unsolvable problem can arise when the substitution results in $T(e) = F(e)$. In this case, the constraint represented by edge $e$ reduces to

$0 \geq CV(e)$,

or

$0 \leq CV(e)$,

depending on $CT(e)$. In this case, the constraint solver checks for consistency. If the edge $e$ represents an inconsistent inequality constraint, it is reported to the user, and the program terminates. If the edge $e$ represents a consistent inequality constraint, then $CT(e)$ is set to $NULL$; the constraint is always satisfied by the equality subgraph.

In the example problem, the inequality graph is now as in **Figure 6**; the equality relationships remain stored in $RV$ and $R$.

• *Phase 3: Initialize for inequalities*

At this point, the entire problem has been reduced to a set of inequality constraints and the equality subgraphs, now reduced to expressing all vertices in terms of each subgraph's representative vertex. The constraint solver must now attack the inequality problem. This it does by another relaxation algorithm, similar to that used for the equality subgraph solution. Prior to this, the solver must initialize various elements. These include externally supplied definitions, and the "assignment level" counter used in trapping cyclic constraint graphs. This initialization proceeds as follows:

*Step 1. Set direct definitions.*

A scan is made of all vertices. For each vertex $v$, the flag $D(v)$ is examined. The following two cases exist:

a. $D(v)$ is $F$.
   No value has been assigned to $X(v)$ either externally (by the user) or internally (by a previous pass of the solver). In this case, set $AL(v) = -1$.
b. $D(v)$ is $T$.
   The value of $X(v)$ is a known value, either by external definition, or as the result of a previous pass of the solver. In this case, set $AL(v) = 0$. If $R(v) = j > 0$, then this defined value must propagate to all other vertices on the same equality subgraph. This propagation is done at this point, with a check for consistency at each vertex.

*Step 2: Propagate definitions.*

At the end of Step 1, most definitions of values have been made and entered into $X(v)$, $D(v)$, and $AL(v)$. However, it is possible that a definition has been made to the representative

vertex of some equality subgraph; Step 1 will not detect this. Step 2 propagates definitions from any representative vertices to all members of the represented equality subgraph. This is done by examining all vertices and specifically looking at $R(v)$. Two cases apply:

a. $R(v) = 0$.
   This vertex is not a member of any subgraph. It is skipped.
b. $R(v) = j > 0$.
   Vertex $v$ is a member of the equality subgraph represented by vertex $j$. If $AL(j) = -1$, this representative vertex has no value, and vertex $v$ is skipped. If $AL(j) = 0$, then vertex $j$ has a value, and, according to the equality subgraph, $X(v) = X(j) + RV(v)$. Following a test for consistency (since vertex $v$ may also be defined) this value is given to vertex $v$, and $AL(v)$ is set to 0.

*Step 3. Rewrite constraints.*
For simplicity in the rest of the algorithm, the rest of the constraints are now all rewritten to be of the "≥" form. This is simply done by examining each edge $e$. If $CT(e)$ signifies ≤, then the constraint is rewritten by transposing $F(e)$ and $T(e)$, negating $CV(e)$, and setting $CT(e)$ to signify "≥".

At the end of Phase 3, the effects of all definitions have been entered into the vertex table, and propagated (where needed) through any equality subgraphs. At this point, the constraint solver begins to operate on the inequality constraints.

• *Phase 4: Inequality solution*
The constraint solver is now ready to attack the inequality problem. This it does by another relaxation algorithm, similar to that used for the equality subgraph solution. In doing this, the solver operates in two "directions." In the "forward" direction, lower bound values propagate from $F(e)$ to $T(e)$; in the "reverse" direction, upper bound values propagate from $T(e)$ to $F(e)$. Either of these processes can involve unterminated loops if inconsistent constraints are contained within the graph. Therefore, an "assignment level" is maintained to intercept these effectively cyclic constraint problems. The details for the "forward" direction follow.

*Step 1. Initialize.*
A counter ($COUNT$) is set to 0. Then a pass is made through all constraints in the edge table. Edge $e$ represents a constraint implying that

$$X[T(e)] \geq X[F(e)] + CV(e).$$

The values of $AL[F(e)]$ and $AL[T(e)]$ are examined, with specific action depending on those values. Specifically, four cases are considered:

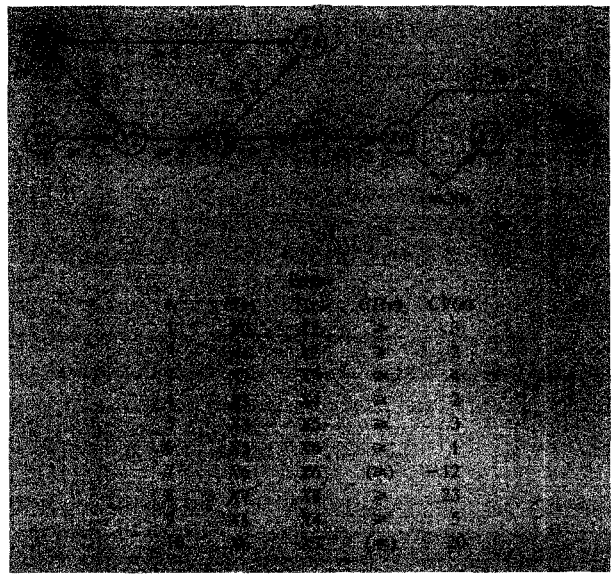

**Figure 6**

Constraint graph after subgraph solution and substitution. Constraint types (CT) in parentheses have become *NULL*; the former type is shown. Constraint 7 is now redundant. Constraint 10 is now contained in the vertex table (Fig. 4).

a. $AL[T(e)] = -1$ and $AL[F(e)] = -1$.
   So far, nothing has given a value to either vertex in this constraint: it is skipped.
b. $AL[T(e)] = -1$ and $AL[F(e)] \geq 0$.
   No lower bound has been given to $T(e)$, but a lower bound is known for $F(e)$. The effect of the constraint is to propagate the value of $F(e)$ to $T(e)$, with a suitable increment. Specifically, set

   $X[T(e)] = X[F(e)] + CV(e)$,
   $AL[T(e)] = AL[F(e)] + 1$,
   $COUNT = COUNT + 1$.

c. $AL[T(e)] = 0$ and $AL[F(e)] = \geq 0$.
   This constraint terminates in a defined vertex. $X[F(e)]$ has a lower bound (or defined value), and $X[T(e)]$ has a defined value. Neither can change if the constraint is not satisfied; therefore a test is made to see that the $X[T(e)]$ is compatible with the constraint and the minimum value of $X[F(e)]$: if so, no action is taken. If not, the error is reported, and the algorithm terminates.
d. $AL[T(e)] \geq 0$ and $AL[F(e)] \geq 0$.
   In this case, both vertices $F(e)$ and $T(e)$ have lower bounds. Since for "forward" propagation lower bounds can only be increased, if constraint $e$ is not currently satisfied, it can only be satisfied by increasing $X[T(e)]$.

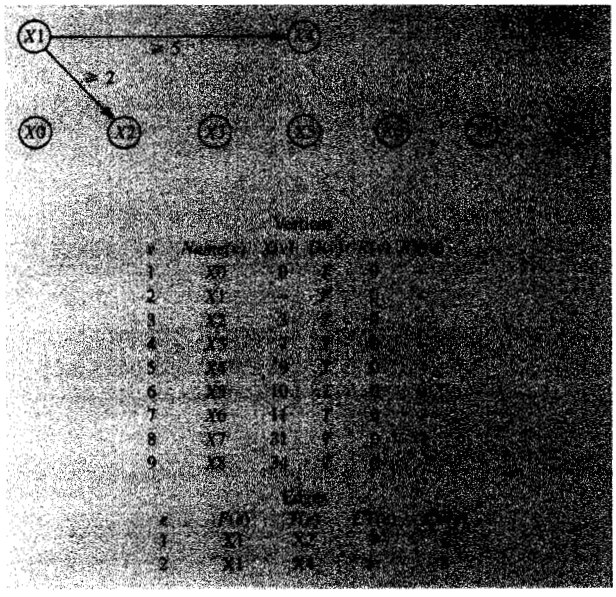

**Figure 7**

Graph after completion of first "forward" pass. Note that $X1$ has not been assigned any value. The "reverse" pass will give this vertex the value 1.

Thus, if $X[T(e)] \geq X[F(e)] + CV(e)$, no action takes place, as the constraint $r$ is satisfied by the present values. If the constraint is not satisfied, the same assignments are made as in (b) above.

At each time a new value is set, $AL[T(e)]$ is tested. If $AL$ exceeds either the total number of unknowns or the total number of constraints in the problem, a cyclic situation exists, and the algorithm stops, reporting a cycle, and displaying all vertices and their current assignment level, to provide the user a preliminary indication of the location of the cycle.

• *Phase 5: Forward slacks*

When a pass of the algorithm takes place which ends with $COUNT = 0$, there are no $\geq$ constraints that have not already been satisfied. All lower bounds for all vertices having $AL > 0$ are at their minimum legal value. As shown in Fig. 2, setting the value of a vertex to its lower bound may result in design-rule-legal but inappropriate shapes. Some vertices must be allowed to be set to their maximum value. In doing this, the constraint solver uses a limited concept of "maximum": a "maximized" vertex will have its value increased only to the point where further increases would require the increase of a nonmaximized vertex value. In other words, "maximize" is equivalent to the removal of slack.

The computation of slack is similar to the basic propagation algorithm given above. Slack is computed for a given vertex $v$ only if three conditions are met:

a. vertex $v$ is to be maximized,
b. $AL(v) > 0$ (vertex $v$ has not yet been permanently fixed but has a lower bound), and
c. there exists an edge $e$ with $F(e) = v$ and $AL[T(e)] \geq 0$.

The last condition prevents the unlimited increase of an otherwise unbounded vertex, such as X8 in the example.

At the end of Phase 5, any vertices $v$ which have been assigned a value $[AL(v) > 0]$ are treated as defined: $AL(v)$ is set to 0, and $D(v)$ is set to $T$. For the example problem this is illustrated in **Figure 7**.

• *Phase 6: Reverse propagation*

One of the desired properties of the constraint solver is that it impose no particular restrictions on the form of the constraint graph. This can lead to considerable complications. Consider the graph of Fig. 3. In this graph, there is no single vertex from which one can reach all other vertices by traversing the edges in their forward direction. If there were, and if it were assumed that the node was to be initially defined to have value 0, the simple forward propagation/slack propagation described above would be a complete algorithm: on terminating (without an error), all vertices would be assigned values consistent with the design rules in the graph, and consistent with the "maximize" concept also described above. It is possible to argue that if the constraints are generated by a program, they can be given this property; it is not possible to ensure that the property is maintained after any manual modification of the constraint graph. To allow for this, the constraint solver next repeats the propagate/slack phase. However, this time the direction of propagation is reversed: upper bound information propagates from vertex $T(e)$ to vertex $F(e)$, and the slack step determines minimum values of vertices. The details follow from the "forward" phases above, and are not repeated.

After the reverse propagation, a test is again made to see if any changes were made in the entire step. If so, the solver returns to Phase 4, to begin more forward propagation. While this alternation between forward and reverse propagation steps is complex, it results in solutions that are in good agreement with designers' expectations regardless of the vertex (or vertices) used as starting points. In fact, the code is not as complex as might be expected, since most of it is shared between the forward and reverse cases.

• *Phase 7: Back substitution*

Once all inequalities have been solved, it remains for the nonrepresentative members of equality subgraphs to be evaluated. This is now simple, since each such vertex carries a pointer to the appropriate representative vertex, and a

value relative to that vertex. If the representative vertex has become defined (as a result of relaxation algorithm above), all vertices that it represents are defined.

• *Phase 8: Clean-up*

At this point, all that remains is to remove any constraints that have become irrelevant. Since a vertex is never altered after it has been marked as defined, all that must be done in clean-up is to delete those constraints that refer to two defined vertices. This is done, and completes the solution algorithm.

## Summary and conclusions

This paper has presented the details of a constraint solver used to solve a general class of mixed inequality/equality constraint relationships for use in a general design-rule-independent mask generation system. The algorithm handles any such problem (within its table sizes) and, where constraints are specified in ambiguous ways, results in solutions that generally agree with designers' expectations. The algorithm is efficient in its use of storage. Written in FORTRAN-IV, it requires 10 bytes of storage per constraint, and 26 bytes of storage per vertex. It is also reasonably fast; typical moderately large problems, with their solution times, appear in **Table 1**. For these problems, which include part or all of a real chip design, there are 3.5–4 constraints per unknown; running time is reasonably well given by

$$t \cong 30.13 \times N_u^{1.472} \text{ (microseconds)},$$

where $N_u$ is the number of unknowns in the problem; quite large problems are not unreasonable. (The largest problem in the table performs the complete design of a memory chip.) Solutions, in addition to meeting constraints, also generally agree with designers' expectations. The algorithm has been in use in a design-rule-independent layout system for approximately two years, and has been found highly effective in the creation of macros suitable for IC design. The principal weakness in the constraint solver is the restricted class of constraints that it can accommodate. This class is clearly useful, since it covers all cases required by typical design rules. However, the addition of a 3-variable class of constraint of the form

$$X_1 + X_3 = 2 \times X_2$$

would be a significant extension since it specifies that $X_1$ and $X_3$ are to be symmetrical about $X_2$. Such symmetry would be very useful in designing electrically balanced circuits (such as sense amplifiers), where electrical balance is usually achieved by providing geometrical symmetry. However, the added class of constraint is significantly more complex than the simpler constraints already accommodated by the constraint solver, particularly if $X_1$, $X_2$, and $X_3$ are allowed to be members of any other constraint without restriction. The possibility of adding this class of constraint is under exploration.

**Table 1** 3081 Running time.

| Problem | | | Time (CPU s) |
|---|---|---|---|
| $N_v$ | $N_u$ | $N_e$ | |
| 1154 | 1029 | 3690 | 0.82 |
| 1554 | 1426 | 5265 | 1.34 |
| 2535 | 2407 | 8955 | 2.81 |
| 18344 | 18162 | 69878 | 56.14 |

$N_v$ = Total vertices in graph.
$N_u$ = Vertices corresponding to unknowns.
$N_e$ = Total edges (constraints) in graph.

## References


1. M.-Y. Hsueh and D. O. Pederson, "Computer Aided Layout of LSI Circuit Building Blocks," *Proceedings of the 1979 International Symposium on Circuits and Systems*, July 1979, pp. 474–477.
2. A. E. Dunlop, "SLIM—The Translation of Symbolic Layout into Mask Data," *Proceedings of the 17th Design Automation Conference*, June 1980, pp. 595–602.
3. N. Weste, "Virtual Grid Symbolic Layout," *Proceedings of the 18th Design Automation Conference*, June 1981, pp. 225–233.
4. K. H. Keller, A. R. Newton, and S. Ellis, "A Symbolic Design System for Integrated Circuits," *Proceedings of the 19th Design Automation Conference*, June 1982, pp. 460–466.
5. R. Lipton, S. North, R. Sedgewick, J. Valdes, and G. Vijayan, "ALI: A Procedural Language to Describe VLSI Layouts," *Proceedings of the 19th Design Automation Conference*, June 1982, pp. 467–475.
6. L.-Z. Liao and C. K. Wong, "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints," *Proceedings of the 20th Design Automation Conference*, June 1983, pp. 107–112.


*Received December 18, 1983; revised April 25, 1984*

**Peter W. Cook** *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Cook attended the University of Cincinnati, Ohio, receiving the B.S. degree in electrical engineering in 1962, and Carnegie-Mellon University, Pittsburgh, Pennsylvania, where he received his M.S. and Ph.D. in electrical engineering in 1968 and 1971. Following graduation from Cincinnati, he was a member of the staff of the Laboratory of Technical Development of the National Heart Institute at the National Institutes of Health in Bethesda, Maryland, where he developed instrumentation for cardiovascular system research. In late 1965, Dr. Cook joined the IBM Thomas J. Watson Research Center. His activities at IBM have centered on aspects of MOSFET LSI/VLSI design, including mask generation, circuit design, and design tools. He is currently manager of the VLSI logic group at the Research Center.