Magic's Incremental Design-Rule Checker

George S. Taylor and John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences Department
University of California
Berkeley, California 94720

ABSTRACT

The Magic VLSI layout editor contains an incremental design-rule checker. When the circuit is changed, only the modified areas are rechecked. The checker runs continuously in background to keep information about design-rule violations up-to-date. This paper describes the basic rule checker, which operates on edges in the layout, and the techniques used to perform incremental checking on hierarchical designs.

Keywords and Phrases: design-rule checking, interactive layout editor

1. Introduction

Almost all existing design-rule checking (DRC) programs are batch oriented [1] [2]. They read in a complete circuit layout and check the entire design. If the circuit is changed, the only way to find out whether design rules have been violated is to recheck the entire design, no matter how small the change or how large the design. For chips with tens of thousands of transistors, batch DRC run may require hours of computer time.

This paper describes a different approach to design-rule checking. As part of the Magic VLSI layout editor [3], we have built a checker that operates *incrementally*. When the layout is modified, Magic records which areas have changed and rechecks only those areas. While the user continues editing, the checker runs in background and highlights errors as it finds them. There is no set-up time because it works from the same data structure used to represent the layout. Since most changes made with the interactive editor are small and the checker is fast, it can usually display errors instantly.

The user's view of design-rule checking is a simple one. As he edits the circuit, small white dots appear over areas that contain layout errors. As soon as the errors are fixed, the white dots go away. Error informaton is stored with the design and it will reappear during the next editing session if the violation has not been fixed. This information is always kept up-to-date, so there is never any need to run a batch checker.

In the next section, we describe Magic's internal representation for a layout and explain how particular features contribute to fast incremental checking. Section 3 describes how the basic checker works from edges in the layout and how design rules are specified. Section 4 shows how we use the basic checker for incremental checking of individual cells, and section 5 describes how hierarchical designs are handled. Section 6 gives measurements of the checker's speed.

2. Representation of a Layout

In Magic, a layout is represented as a hierarchical collection of cells. Each cell contains mask information plus pointers to subcells. For now, we will consider only a single cell at a time (Section 5 generalizes the solution to handle hierarchical designs).

Magic represents the mask layers of a cell with rectangular *tiles*, which means that it handles only Manhattan geometries. Each tile indicates the type of mask layer it represents. Tiles are connected to form *planes* by a technique called *corner-stitching* [2] illustrated in Figure 1. The tiles in a plane are non-overlapping and cover it completely. Empty areas are covered with tiles of type "space."

Each cell contains several planes of mask information. Mask types that interact (such as polysilicon and diffusion) are stored together in the same

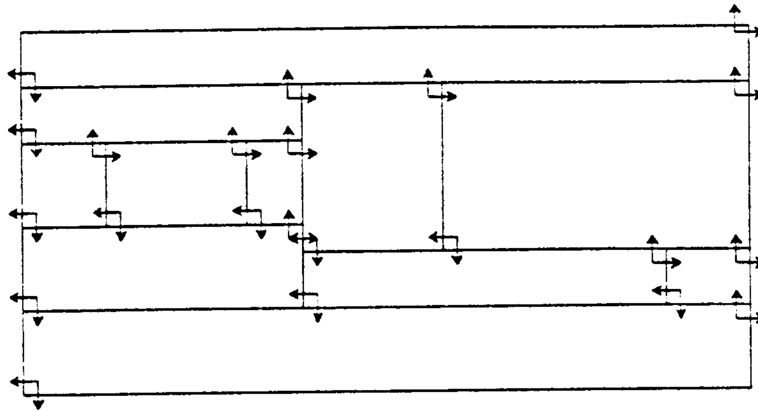

Figure 1. An example of a corner-stitched plane. Each plane contains tiles of different types that cover the entire area of the plane (space tiles are used where there is no mask material). Each tile contains four pointers that link it to neighboring tiles at its corners. The pointers make it easy to find all the material in a given area.

plane, while those that do not interact (such as polysilicon and metal) are stored in different planes. Contacts between mask types on different planes are represented in both of them. Our nMOS process has two planes: one for metal and one for polysilicon, diffusion, and transistors.

Instead of working directly with physical mask layers, Magic uses *abstract layers* to represent structures such as transistors and contacts. The abstract layers appear in the database as tiles with special types. For example, instead of representing an enhancement transistor as a polysilicon tile over a diffusion tile, it is represented with a tile of type "enhancement transistor." A more complete explanation of the abstract layers is given in [3]. What matters here is that all the interesting features are represented explicitly: there is no need to cross-register diffusion and polysilicon to discover the transistors.

The design-rule checker takes advantage of Magic's database in three ways. First, the corner-stitched tiles allow DRC to find material in a given area very quickly. Second, division of mask information into planes allows the checker to work with one plane at a time, ignoring irrelevant geometry on other planes. Third, there is no need to extract features by registering layers: the abstract layers represent the important features explicitly. Because of these features, there is no need for the checker to manage a separate structure of its own: it works directly from the layout database.

3. The Basic Checker

This section describes the basic design-rule checking paradigm used to validate an area of a single corner-stitched plane. Later sections show how this basic checker is used to perform incremental checks on a single cell, and then on a hierarchy of cells.

3.1. Edge-based Rules

Magic's design rules are based on edges between tiles. Each rule can be applied in any of four directions, two for horizontal edges and two for vertical edges. The rule database contains a separate list of rules for each possible combination of materials on the first and second sides of an edge. In its simplest form, a rule specifies a distance and a set of mask types: only the given types are permitted within that distance on the second side of the edge. This

area is referred to as the *constraint region*. See Figure 2.

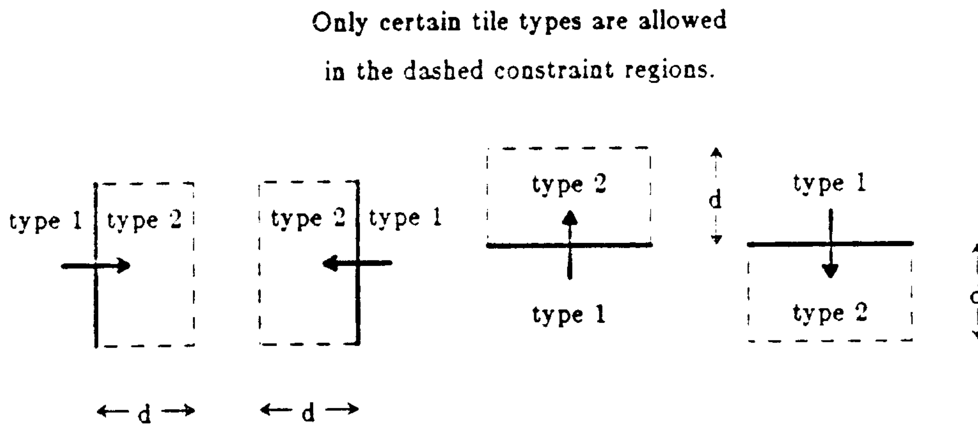

Figure 2. Design rules are applied at the edges between tiles in the same plane. A rule is specified in terms of *type 1* and *type 2*, the materials on either side of the edge. Each rule may be applied in any of four directions, as shown by the arrows. The simplest rules require that only certain mask types can appear within distance *d* on *type 2*'s side of the edge.

Unfortunately, this simple scheme will miss errors in corner regions, as shown in Figure 3. To eliminate these problems, the full rule format allows the constraint region to be extended past the ends of the edge under some circumstances. See Figure 4 for an illustration of the corner rules and how they work. Table 1 gives a complete summary of the information in each design rule.

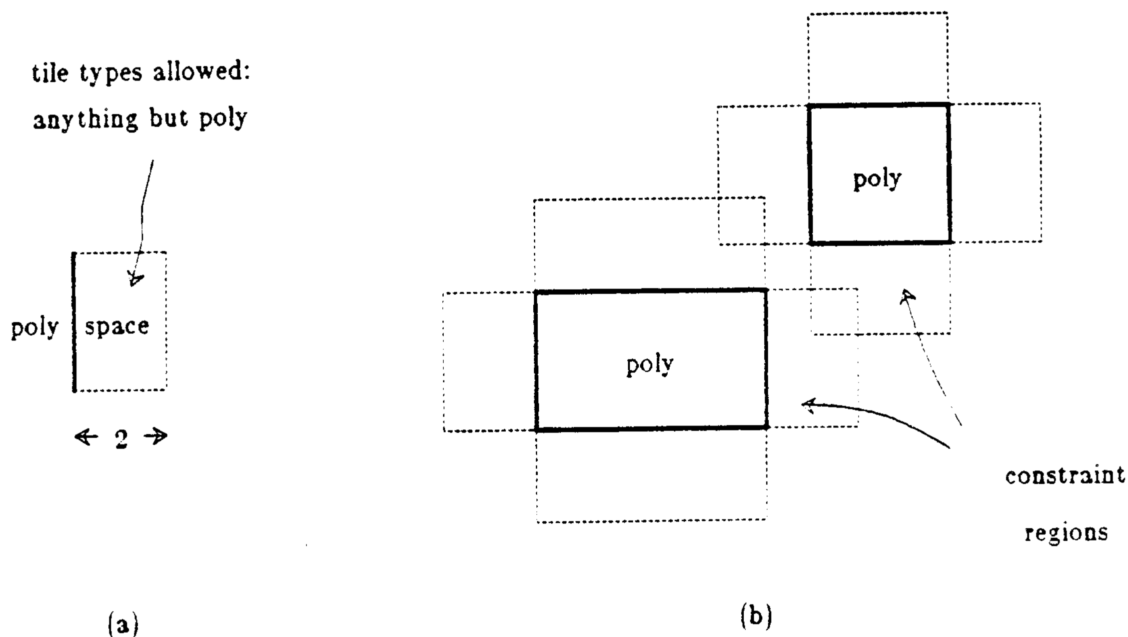

Figure 3. If only the simple rules from Figure 2 are used, errors may go unnoticed in corner regions. For example, the polysilicon spacing rule in (a) will fail to detect the error in (b).

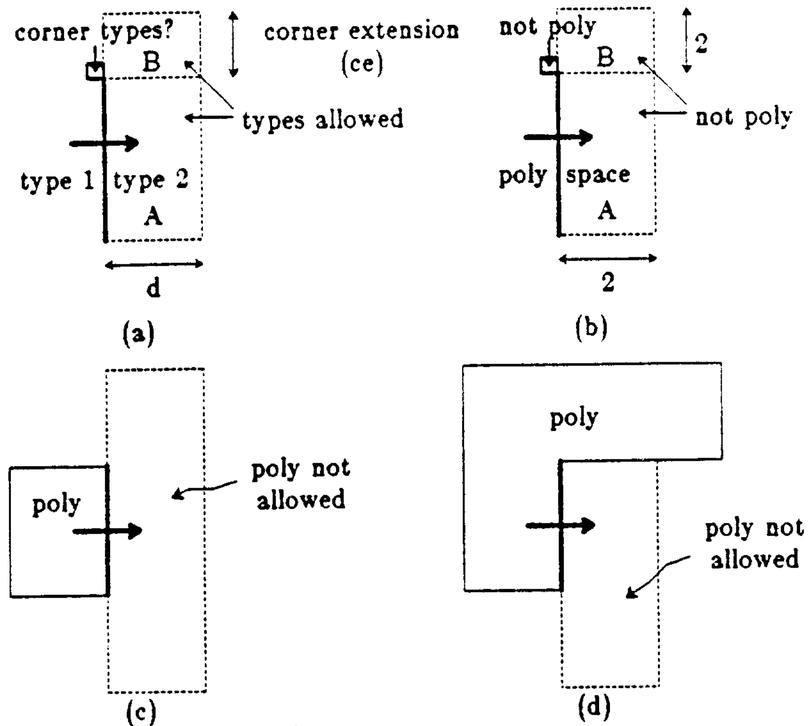

Figure 4. The complete design rule format is illustrated in (a). Whenever an edge has *type 1* on its left side and *type 2* on its right side, the area A is checked to be sure that only *types allowed* are present. If the material just above and to the left of the edge is one of *corner types*, then area B is also checked to be sure that it contains only *types allowed*. A similar corner check is made at the bottom of the edge. Figure (b) shows one of the polysilicon spacing rules, (c) shows a situation where corner extension is performed on both ends of the edge, and (d) shows a situation where corner extension is made only at the bottom of the edge.

Parameter	Meaning
type 1	Material on first side of edge.
type 2	Material on second side of edge.
d	Distance to check on second side of edge.
layers allowed	List of layers that are permitted within d units on second side of edge.
corner types	List of layers that cause corner extension.
ce	Amount to extend constraint area when corner types match.

Table 1. The parts of an edge-based rule.

3.2. Applying the Rules

To check an area of a single plane, Magic must first find all the edges in that area. This is accomplished by searching for all the tiles in the area. The corner-stitched data structure is well suited to searches of this sort: see [4]. For each tile, the checker examines its left and bottom sides (the top and right sides of the tile will be checked by the neighbors on those sides). Since the tile may have neighbors of different types on the same side, the checker searches through all the neighbors to divide the side of the tile into edges with a single material on each side.

To process an edge, the mask types on each side of it are used to index into the rule table to find the list of rules for that kind of edge. Each rule in the list is checked, and white dots are displayed for any areas where the constraints are not satisfied. For each edge there are two rule applications: left-to-right and right-to-left (for vertical edges) or bottom-to-top and top-to-bottom (for horizontal edges). A different list of rules is applied in each direction, since the layers are reversed.

3.3. Specifying Design Rules

Design rules are specified in a technology file that contains the rules and other technology-specific information. When Magic starts executing, it reads this file and builds the rule table. Initially we specified rules in the detailed form of Table I, with one line for each edge rule. This scheme proved to be

unworkable, because there were many rules and it became difficult to convince ourselves that the rule set was complete and correct.

In order to simplify the process of creating rule sets, Magic now permits rules to be specified with high level macros for width and spacing. For example, the macro

spacing ef DP 1

is expanded into several rules to verify that types e and f (enhancement and depletion transistors) are always separated from types D and P (diffusion-metal contacts and poly-metal contacts) by at least one unit. The macro

width pPBef 2

is expanded into the set of edge rules needed to verify that the entire region containing any of the five types P, B, e, f or p (polysilicon) is always at least two units wide.

Most of the rules for our processes are simple width and spacing checks, so these two macros considerably simplify the writing of rule sets. Our nMOS rule set contains 8 width rules, 6 spacing rules, and 9 of the detailed edge rules for situations that cannot be handled by the width and spacing rules (e.g. transistor overhangs). Magic expands these 23 high-level rules into 126 detailed edge rules. The complete high-level rule set for nMOS is given in the Appendix.

The width and spacing macros make Magic's checker more efficient because the width and spacing rules are symmetric. If layers x and y are too close together, the violation can be detected from either an edge of x or an edge of y. This means that it is unnecessary to check the rules from both edges. Magic takes advantage of this symmetry by checking width and spacing rules in only two directions (left-to-right and bottom-to-top). In addition, symmetric rules mean that corner extension is only necessary on one end of each edge. Since most of the detailed edge rules come from the width and spacing macros, this speeds up the checking process by almost a factor of two.

4. Continuous Design-Rule Checking

This section shows how the basic checker is used to provide continuous incremental rule validation. As in the previous section, we consider only single-cell designs here.

In order to perform DRC incrementally, Magic maintains two extra kinds of information with each cell, stored in the same form as mask layers. First, Magic keeps information about rule violations that have been detected but haven't been corrected. The violations are represented by error tiles that cover the areas where rule constraints are not satisfied. The second kind of information consists of tiles describing the areas of the circuit that need to be reverified. The error tiles and the reverify tiles are stored in separate corner-

stitched planes. Each cell contains its own error and reverify planes.

When a designer changes a cell, Magic creates reverify tiles that cover the area modified. The design-rule checker runs in background while Magic is waiting for the designer to enter the next command. DRC first searches for reverify tiles. Then it invokes the basic checker over the area covered by each tile found. The basic checker reverifies the area on each of the cell's planes, updates error tiles, and erases the reverify tile. Changes to the error information are reflected immediately on the graphics screen.

If the designer invokes a command while the checker is running, the checker stops so that the command can be processed without delay. After the command finishes, the checker resumes by starting over on the area that it was working on just before the interruption. Large reverify tiles are broken up into small ones before checking, in order to reduce the amount of work that might have to be repeated. When there are large areas to be reverified, the checker works across the design in a style like "Pac-Man," gobbling up reverify tiles and spitting out error tiles.

If incremental checking is done carelessly, errors may not be detected when new violations are introduced, and error information may be left in the database even after the violations have been corrected. Figure 5 illustrates the problem and Magic's solution. When an area is modified, error information may be affected in both the area that was modified and in the surrounding

area (for example, material in area A may be too close to something in the surrounding area B). We call the surrounding area the *halo*. Its width is equal to the largest distance in any design rule. Error information must be recomputed in the modified area and its halo. However, errors in the halo don't necessarily involve the inner modified area. They may come from interactions between the halo and a second halo outside it. To regenerate errors in the first halo correctly, information in the second halo must be considered.

If area A of Figure 5 were modified, Magic would recheck it by deleting all error information in A and B. The checker would then generate new error information in both areas by invoking the basic checker over areas A, B and C. Any errors found during this process would be clipped to the area of A and B, so that error information outside the region where errors were erased would not be affected.

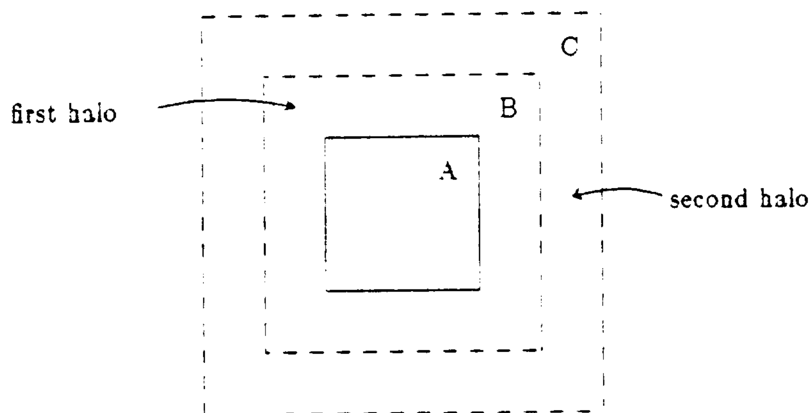

Figure 5. If area A is modified, the design-rule checker erases existing error information in both A and B. Errors in B could have come from information in A, B or C, so all three areas must be checked to regenerate all of the errors. The width of the *halos* B and C is equal to the largest distance in any design rule.

The reverify and error tiles are stored with cells so that they are not lost at the end of an editing session. Normally, there will be no reverify tiles left at the end of a session, but if a large area has been changed recently, it is possible that it won't have been reverified when the session ends. In this case, the reverify tiles are written to disk with the cell. When the cell is read in during the next editing session, the design-rule checker will notice the reverify tiles and continue the reverification process. The reverify and error tiles are identical to the tiles used to represent mask layers, except that they are not manipulated directly by the designer.

5. Hierarchical Checking

Most of the layouts created with Magic consist of hierarchical cell structures rather than single cells (Figure 6). Each cell may contain subcells, and the subcells may overlap other subcells or mask information in the parent. A subcell may be appear any number of times in any number of parents.

In hierarchical designs, errors can arise in any of three ways:

a) the mask information of an individual cell may be incorrect;

b) a subcell may interact incorrectly with another subcell; and

c) a subcell may interact incorrectly with mask information in its parents.

Magic's incremental checker includes facilities to detect all of these errors. Overlapping subcells are no more difficult to handle than subcells that merely

abut, because interaction errors are possible in either case.

5.1. Simple Checks and Interaction Checks

Two overall rules guide the hierarchical checker. First, the mask information in every cell is required to satisfy the design rules by itself, without consideration of subcells. Second, each cell and its subcells must together satisfy all the design rules, without consideration of how that cell is used in its parents. If the layout is viewed as a tree structure, the first rule means that each node of the tree must be consistent, and the second rule means that each subtree must be consistent.

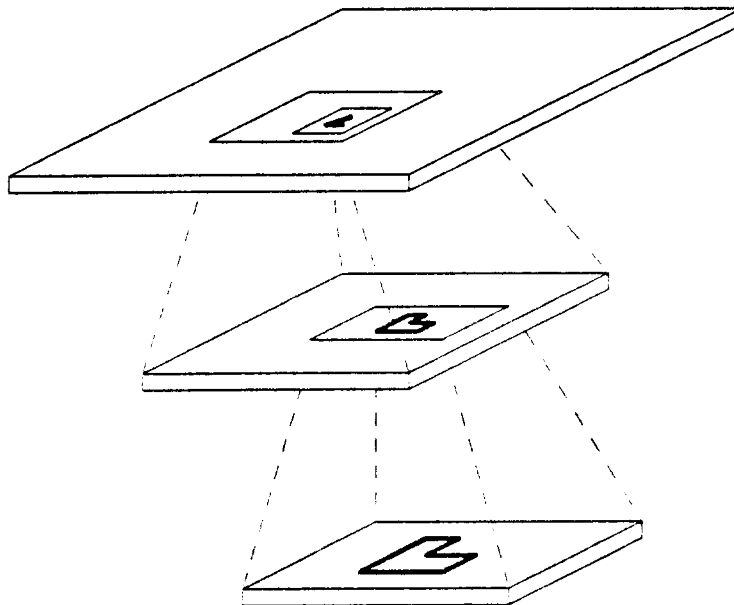

Figure 6. Circuits are defined by cells arranged in a hierarchy. If mask information is changed in a low-level cell, Magic checks to be sure that the cell is consistent by itself and that there are no illegal interactions in parents or grandparents.

The overall rules result in two kinds of design-rule checking. The first rule is verified by running the basic checker over the planes containing mask information for each cell; this is called a *simple check*. The second rule is verified with an *interaction check* which considers interactions involving subcells. Each cell uses separate planes to hold its mask information, so interaction checks must combine information from different planes.

To make an interaction check on an area, the hierarchical structure is "flattened" to produce a new set of corner-stitched planes that combines all the information from all cells in the area to be checked. This includes mask information from the parent cell, plus mask information from subcells and sub-subcells, and so on. Once all the mask information in the area has been collected into a single set of planes, the basic checker is invoked on these planes in the standard fashion (halo expansion is performed as described in Section 4). Errors arising from the interaction check are placed in the parent cell.

Interaction checks are more expensive than basic checks, since they involve flattening a piece of the hierarchy. Fortunately, interaction checks can often be avoided. For example, if an area contains no subcells, then there is no need to perform an interaction check on that area. A simple check will find all errors. The interaction check can also be avoided if there is only a single subcell in an area, with no other subcells or mask information nearby. In

this case any errors must come from within the subcell, and those errors will be found by checks made within that cell. Interaction checks are necessary only in areas where a subcell is within one halo distance of mask information or another subcell. Even then, we only need to check the the area around the interaction.

5.2. Checking Upward in the Hierarchy

When a cell is modified, simple checks and interaction checks have to be performed within that cell, and also within its parents in the hierarchy. For example, suppose mask information has been edited within a cell. Then a simple check must be performed within that cell, as well as an interaction check if there are subcells near the modified area. However, these two checks are not sufficient. If the modified cell is a subcell of other higher-level cells, then the change may have introduced interaction problems within the higher-level cells. For each parent of the modified cell, an interaction check must be performed over the area of the modification. Interaction checks must also be performed in grandparents, and so-on up to the top-level cell in the hierarchy. In the cell that was modified, both simple and interaction checks must be performed, but in the parents and grandparents only interaction checks are necessary.

Magic uses two kinds of verify tiles to handle the two kinds of checks. When a cell is modified, "verify-all" tiles are placed in that cell to signify that both simple and interaction checks must be performed. At the same time,

"verify-interactions" tiles are placed in parents and grandparents to indicate that interaction checks have to be performed. The background checker keeps track of which cells in the database contain verify tiles and performs each kind of check wherever necessary.

In the worst case, the hierarchical algorithm could result in the modified area being rechecked once at each level of the hierarchy above the cell that was changed, with a separate flatten operation required for each check. However, in deep hierarchies most of the interaction checks are avoidable: in cells far above the modified one, the modified area will almost certainly appear in the middle of a single subcell with no mask information or other subcells nearby. Unless there are many large subcell overlaps, any given area of mask information is likely to require an interaction check at only one point in the hierarchy.

5.3. Arrays

One other form of hierarchical check arises because Magic has an array construct. To simplify the creation of cell arrays, Magic contains a special array facility: each subcell may consist of either a single instance or a one- or two-dimensional array of identical instances. Because of the array construct, there is actually a third overall rule that guides the hierarchical checker: each array must satisfy all the design rules, independently of other information in the parent containing the array. Whenever a change is made to an array, the

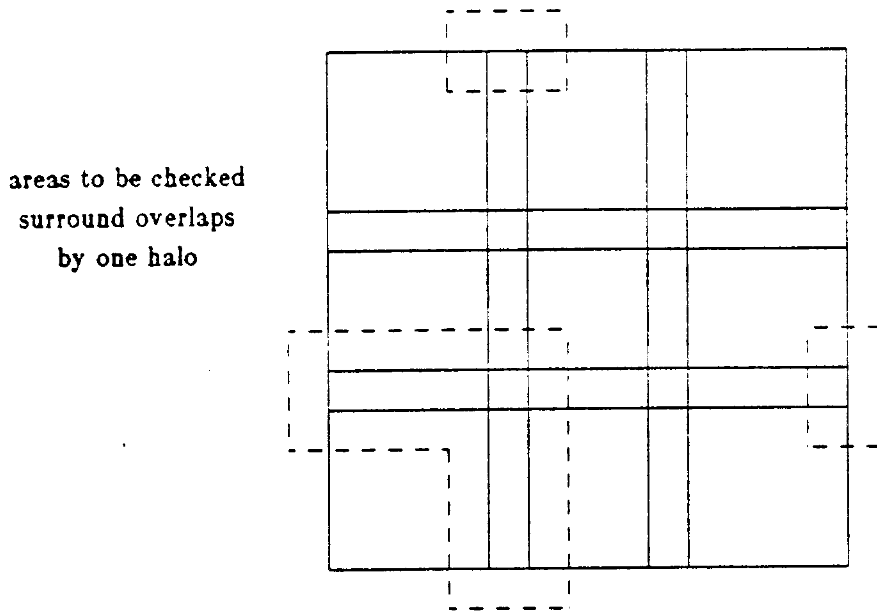

Figure 7. An array is internally consistent if the three dotted areas satisfy the design rules. All possible interactions between elements of the array are identical to the ones that occur these three regions.

array structure is reverified by checking the three areas shown in Figure 7.

6. Implementation and Performance

The design-rule checker is written in C. Its 2000 lines of code are divided into roughly equal thirds for building the internal rule table from the technology file, implementing the basic checker on one plane, and providing for hierarchical checking.

The incremental checking system has just recently become operational. We've made preliminary measurements on single cells with the untuned system. The basic checker processes 200 tiles per second on a VAX 11/780 running Unix. To compare Magic's performance with that of other systems, we

state their speeds in terms of transistors checked per second in Table 2.

A typical change to a circuit involves only a few tiles, so the cost of incremental reverification is dominated by the size of the halos. From this, we estimate that roughly 50 tiles have to be checked per command in an nMOS design. This requires about one-fourth of a second of CPU time.

The average number of edges found per tile is 2.5, but only 1.8 of these have different mask types on the two sides of the edge. An average of 1.7 rules are applied per non-trivial edge.

7. Conclusions

Magic's design-rule checker demonstrates that incremental checking is feasible. We think that circuit designers will find that continuous feedback reduces the time needed to create new designs or modify existing ones. The key to the incremental checker is low overhead: the ability to run from the same database as the interactive editor, the ability to find important edges in the layout quickly, and the ability to find nearby material quickly. The two

System	Transistors / second
Lyra [2]	2
Baker [1]	3
Mart [5]	6-8
Magic	10-15

Table 2. Performance of several design rule checkers. All of the programs were run on a VAX 11/780.

features of Magic's database that reduce overhead are the corner-stitched tile planes and the abstract mask layers. Extending the checker to work in hierarchical designs frees the designer from tedious reverification of interactions when subcells are revised.

8. Acknowledgements

Gordon Hamachi, Bob Mayo, and Walter Scott all participated in discussions that led to the incremental checker and provided many useful comments on drafts of this paper.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD), under Contract No. N00034-K-0251.

9. References

[1] C. M. Baker and C. Terman, "Tools for verifying integrated circuit designs," *Lambda* (now *VLSI Design*) Vol. 1, No. 3 (1980), pp. 22-30.

[2] M. H. Arnold and J. K. Ousterhout, "Lyra: A New Approach to Geometric Layout Rule Checking," *Proc. 19th Design Automation Conference*, June, 1982, pp. 530-36.

[3] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott and G. S. Taylor, "Magic: A VLSI Layout System," included in this technical report.

[4] J. K. Ousterhout, "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools," Technical Report UCB/CSD 82/114, Computer Science Division, University of California, Berkeley, December, 1982. To appear in *IEEE Transactions on CAD/ICAS*, January, 1984.

[5] B. J. Nelson and M. A. Shand, "An Integrated, Technology Independent, High Performance Artwork Analyzer for VLSI Circuit Design," Technical Report VLSI-TR-83-4-1, VLSI Program, Division of Computing Research, CSIRO, Eastwood, SA 5063, Australia, April, 1983.

10. Appendix

To illustrate how Magic is programmed for a particular technology, this section lists the design rules for an nMOS process with buried contacts and a single level of metal. Most rules are specified using width and spacing macros which Magic expands into detailed lower-level rules. Detailed edge and four-way rules may also be specified directly. Table 3 gives the abbreviations that we use for the names of mask types.

Plane	Abbreviation	Mask type
Poly/Diffusion plane:	s	space
	d	diffusion
	p	polysilicon
	D	diffusion-metal contact
	P	polysilicon-metal contact
	B	buried contact
	e	enhancement transistor
	f	depletion transistor
Metal plane:	s	space
	m	metal
	X	metal-diffusion contact
	Y	metal-polysilicon contact

Table 3. Single letter abbreviations for the names of mask types.

The rules in Table 4a define minimum line widths and feature sizes. The first three rules are for the line widths of diffusion, metal and polysilicon. The last five rules define the sizes of contacts and transistors. The *types* field may include one or more mask types. Magic creates a detailed edge rule for all combinations of one member of the *types* field, and one of the mask types in the same plane that is not included in the *types* field.

	types	d	reason
width	dDBef	2	*diffusion*
width	pPBef	2	*polysilicon*
width	mXY	3	*metal*
width	D	4	*diff/metal contact*
width	P	4	*poly/metal contact*
width	B	2	*buried contact*
width	e	2	*efet*
width	f	2	*dfet*

Table 4a. Width rules.

Table 4b contains spacing rules. We distinguish between spacing rules for types that can never be adjacent and spacing rules that apply only when two pieces of material are separated. In either case, Magic creates a number of detailed edge rules in a manner similar to that for width rules.

The width and spacing macros can be used to specify most symmetrical constraints for a particular technology. The detailed edge rules created from the width and spacing macros are applied only from left-to-right across

	types 1	types 2	d	can be adjacent?	reason
spacing	ef	DP	1	no	*transistor – contact*
spacing	e	f	3	no	*efet – dfet*
spacing	B	e	3	no	*buried contact – efet*
spacing	dDBef	dDBef	3	yes	*diff – diff*
spacing	pPBef	pPBef	2	yes	*poly – poly*
spacing	mXY	mXY	3	yes	*metal – metal*

Table 4b. Spacing rules.

vertical edges in the layout, and from bottom-to-top across horizontal edges. These edge rules always check one corner, also.

To specify asymmetrical constraints and constraints that apply alongside edges but not in corners, we use the explicit edge and fourway rules listed in Table 4c. The fourway rules are applied in both directions across all edges in the layout. They also trigger corner checks on both ends of every edge. The edge rules in Table 4c are similar to the ones derived from the width and spacing macros, but could not be written conveniently in either of those forms.

	type 1	type 2	d	layers allowed	corner types	ce	reason
edge	d	spP	1	s	spP	1	diff – poly spacing
edge	p	sdD	1	s	sdD	1	diff – poly spacing
edge	D	sp	1	s	sp	1	diff – poly spacing
edge	P	sd	1	s	sd	1	diff – poly spacing
fourway	ef	s	1	0	0	0	trans can't touch space
fourway	B	dD	4	sdpDPBf	sdpDPBef	3	b_c – e/et spacing
fourway	f	B	3	B	0	0	b_c next to d/et must be 3x?
fourway	ef	p	2	pP	p	2	poly overhang transistor
fourway	ef	d	2	dD	d	2	diff overhang transistor

Table 4c. Edge and fourway rules.