Institut für Technische Informatik

Abteilung Rechnerarchitektur

Universität Stuttgart
Pfaffenwaldring 47
D-70569 Stuttgart

Studienarbeit Nr. 2334

Simulation of Realistic Defects for Validating Test- and Diagnosis-Algorithms

Hossam El Atali

Studiengang:	GUC Austausch
Prüfer:	Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich
Betreuer:	Dipl.-Inf. Stefan Holst
begonnen am:	05.04.2011
beendet am:	31.08.2011
CR-Klassifikation:	B.7.3

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, 30/08/2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Testing is very important in the manufacture of Integrated Circuits (ICs). This is due to the decrease in technology size which has led to the manufactured circuits being more vulnerable to defects, such as bridging defects. During production, the manufactured circuits are tested by applying test patterns to them. The output of these tests is compared to the expected output and if the two differ, then diagnosis is performed. Diagnosis aims to detect and localize faults and obtain information about them and many diagnosis algorithms exist for that purpose. The information obtained from diagnosis can then be used to alter the design in order to reduce the chances of these faults. The used diagnosis algorithms, however, apply heuristics and therefore must be evaluated with realistic test cases to determine their efficiency.

1.2 Goal

In order to evaluate these algorithms, faults are injected into circuits and the diagnosis algorithm then executed in order to see if the injected faults are localized. Two type of faults which can be injected are stuck-at faults and bridging faults. These two result in coverage of the majority of defects that occur in circuits. For stuck-at faults, exhaustive fault injection of all the nodes in the circuit is possible. For bridging faults, however, the number of combinations of bridged nodes for large circuits is too high and cannot

be tested exhaustively. Hence, selective fault injection is necessary and the goal of this thesis is to obtain a realistic set of bridging faults to inject for evaluation of the diagnosis algorithm presented in [1] and [2].

The diagnosis algorithm under evaluation is implemented in the ADAMA tool. ADAMA supports several fault models, including several bridging fault model variants, as well as random fault injection. However, random fault injection does not provide a realistic set of likely faults for a circuit since it does not take into account the probability of each fault. Therefore, the objective of this thesis is further specified as Inductive Fault Analysis (IFA), which is the process of extracting a list of possible bridging faults from the physical layout of any arbitrary circuit using defect data such as possible defect size and defect size probability, and then using the results of IFA to perform simulation of realistic defects with the final purpose of validating the algorithm. There are, however, certain limitations on the circuit from which the fault list is to be extracted and they are presented throughout this document. It is also important to note that the extracted fault list is fault model independent, meaning that, even though the faults are bridging faults, they are not necessarily wired-AND or wired-OR faults, for example. The fault list simply gives a list of nodes which are likely to be bridged by a defect; which model to use to inject these faults is for the user to decide.

There are already several tools available for IFA. However, only one of them, CARAFE [3], supports the extraction of multi-node bridges. Furthermore, CARAFE is only valid for small layouts. The algorithm which was chosen to be implemented is the one presented in [4]. It supports the extraction of multi-node bridges and is considerably faster than CARAFE. In this thesis, only the interlayer bridge extraction part of the algorithm is implemented. The differences between the tools and the algorithms behind them are explained in more detail in Section 2.3.

It is worth noting that extraction of multi-node bridges is important even if the final target of the user is two-node bridges. This is because the presence of multi-node bridges will significantly affect the ranking of faults and will change the order of the two-node bridging faults. The reason for this is that the extraction of two-node bridges overestimates the WCA for a pair of nodes by adding the WCA for any multi-node bridge involving these two nodes to the two-node WCA of these two nodes [4]. This overestimation of WCA can result in pessimistic yield estimations and designs which are too conservative.

1.3 Overview

The structure of this thesis is as follows. First, background material will be given in the remainder of this chapter. This includes the difference between defects and faults and how they relate to each other. An description of the GDS file format is also included. Several examples of defects and fault models are also given. In Section 2, an explanation of IFA is provided as well as a literature study on the various tools of IFA and an explanation of the chosen algorithm. Section 3 provides the details of the implementation of the chosen IFA algorithm. The results of the algorithm and the evaluation of ADAMAs diagnosis algorithm are presented in Section 4. Finally, a conclusion is given in Section 5.

1.4 Defects

A defect is a deformation in the fabricated layout of the circuit, causing a discrepancy between the desired layout and the fabricated layout. This discrepancy can lead to an error, which is a faulty logical output of the circuit. Defects are generally split into two categories: global defects, which affect a large area of the chip and are easily observed during the early stages of fabrication, and spot defects, which affect a small area of the chip and are harder to observe. Examples of global defects include cracks in the wafer and photolithographic mask misalignments [5]. The main concern of this thesis, however, is spot defects since they are not identified during the early stages of fabrication. They can be further categorized into random spot defects and systematic spot defects.

1.4.1 Random Spot Defects

Random spot defects have a random location and are caused by disturbances in local processes occurring during fabrication [6]. Such disturbances include dust particles or other droplets which are unintentionally deposited on the surface of the wafer during critical stages of the fabrication, such as photolithography. These disturbances can then cause shorts or breaks in conducting or insulating layers which ultimately affect the operation of the chip.

Some examples of these disturbances are shown in Figures 1.1 - 1.6 [1] .

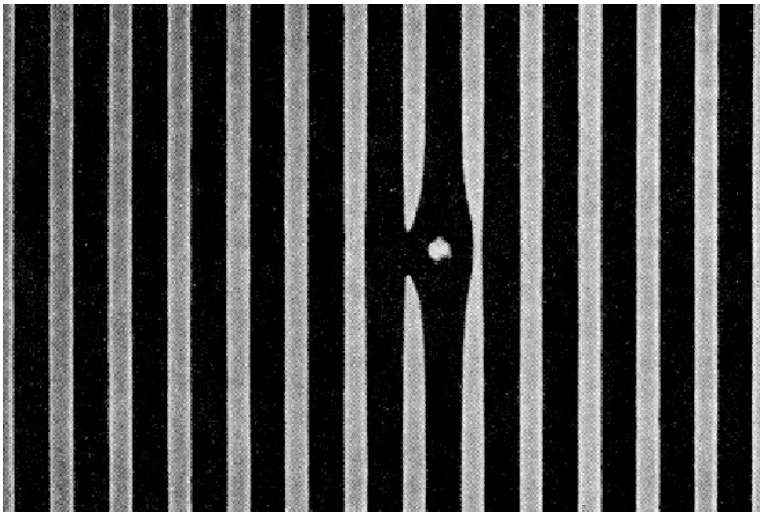


Figure 1.1: Example of a break in a metal layer

In Figure 1.1 and 1.2, contaminating particles cause enlargements of the photoresist which lead to breaks in the metal layer. In Figure 1.3, a contaminating opaque particle falls on the photolithography mask preventing a part of the photo resist from being exposed hence preventing a part of the metal layer from being etched. This results in a short between the metal lines. In Figure 1.4, the contaminating particle is deposited on the metal layer itself instead of the mask. When the photo resist is then deposited, it becomes thicker around the contaminating particle than it should be, which, again, prevents a part of the metal layer from being etched and, again, results in a short between the lines.

Figure 1.5 shows a defect due to a scratch in the photoresist which cannot be classified as either a short or a break since it has several effects. Figure 1.6 shows a rare defect which occurs due to corrosion of the metal.

[1] These figures are taken from [6]. Permission to copy these figures was granted by the Association for Computing Machinery (ACM) as per their copyright notice in the cited paper. Further copying is by permission of the ACM.

Figure 1.2: Example of a large break in a metal layer

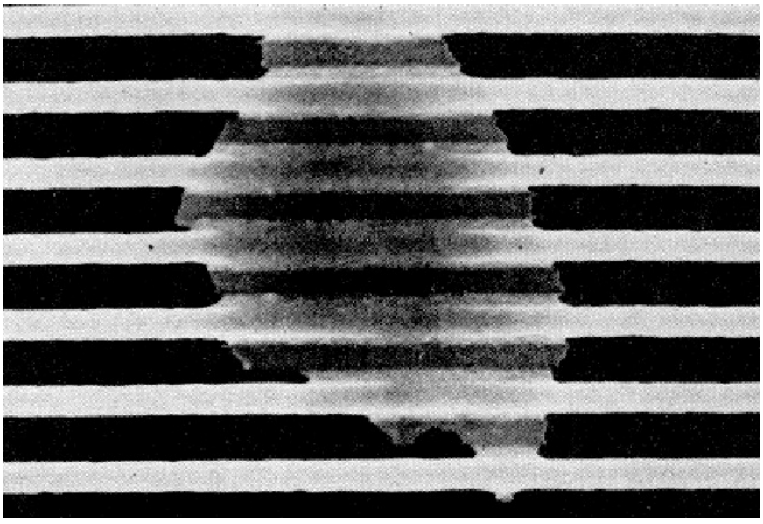


Figure 1.3: Example of a short between metal lines

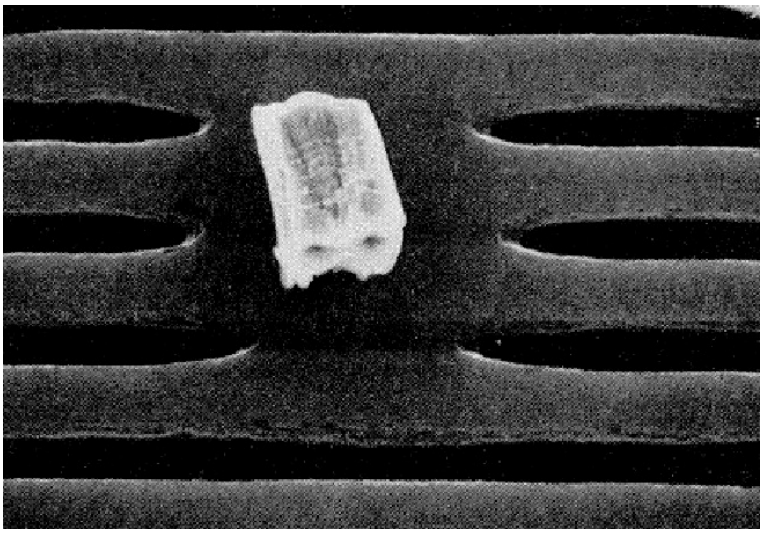


Figure 1.4: Another example of a short between metal lines

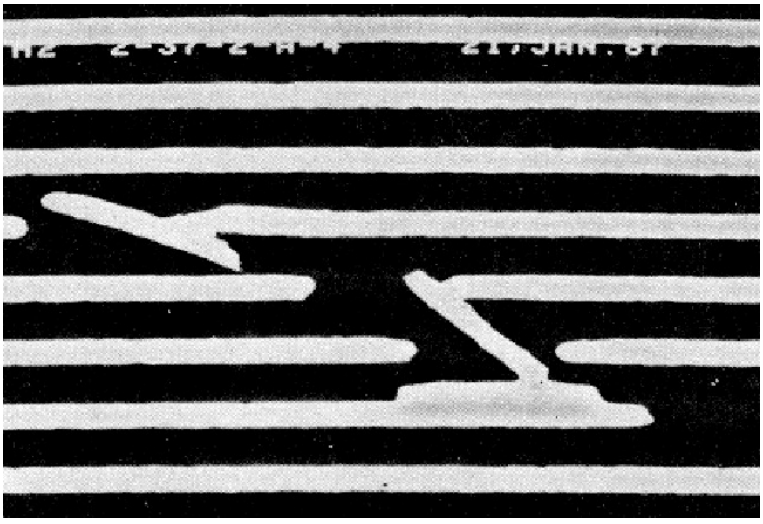


Figure 1.5: Example of a defect resulting in various shorts and breaks

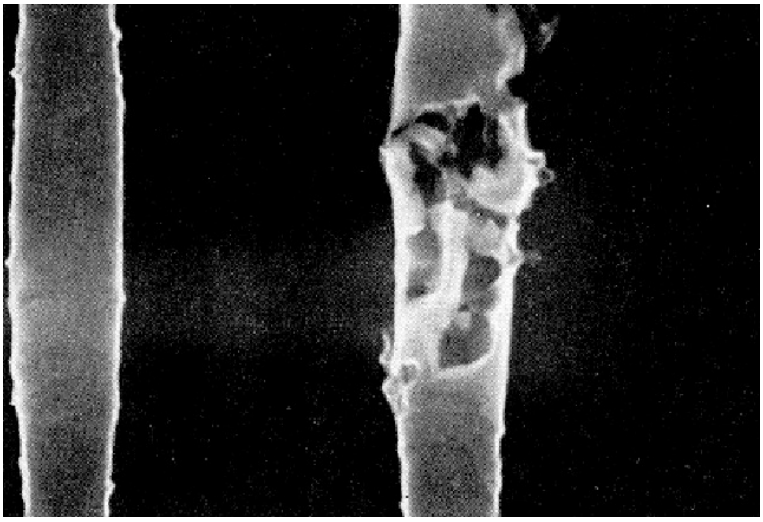


Figure 1.6: Example of a defect caused by metal corrosion

1.4.2 Systematic Spot Defects

Systematic spot defects occur when certain patterns in the layout cause problems during fabrication. Many of these problems are lithography-related. Below, some examples of systematic spot defects are shown. The kind of defect which can occur depends on the lithography process used. For example, optical photolithography normally uses a light source having a wavelength of 193nm [7] to produce features with a much shorter length. This makes diffraction a significant problem which can lead to distortion of the layout if not dealt with. Line shortening, forbidden pitches and corner rounding are common problems associated with 193nm lithography which can lead to opens in metal lines [8]. On the other hand, electron beam lithography (EBL) is much more resistant to diffraction due to the much shorter wavelength of the electron beam. However, it suffers from other problems, such as the proximity effect, which can also lead to distortion of the layout.

Electron Beam Proximity Effect

This form of defect occurs when the electrons aimed at the photoresist penetrate the photoresist and then are scattered by the atoms in the substrate.

This causes the electrons to reflect and affect a part of the photoresist which was not meant to receive a dose of the electrons. The final result of this is a wider pattern on the chip than is intended. This effect can lead to shorts between metal lines which are close to each other. An illustration of this effect is shown in Figure 1.7.

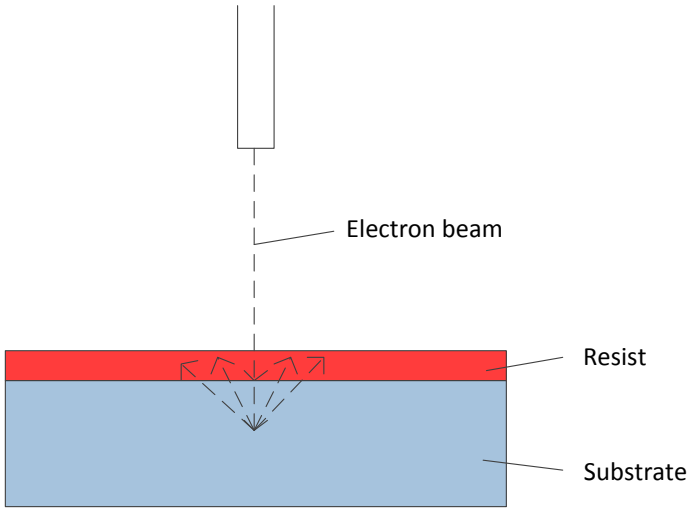


Figure 1.7: The proximity effect in EBL

Forbidden Pitches

This form of defect occurs when certain separation distances between metal lines result in these metal lines becoming very constricted or even broken during fabrication. These separation distances are referred to as "forbidden pitches". An example of this effect is shown in Figure 1.8.

Line Shortening

This defect is when a certain wire in the layout becomes shorter than intended after fabrication. An example of this is shown in Figure 1.9.

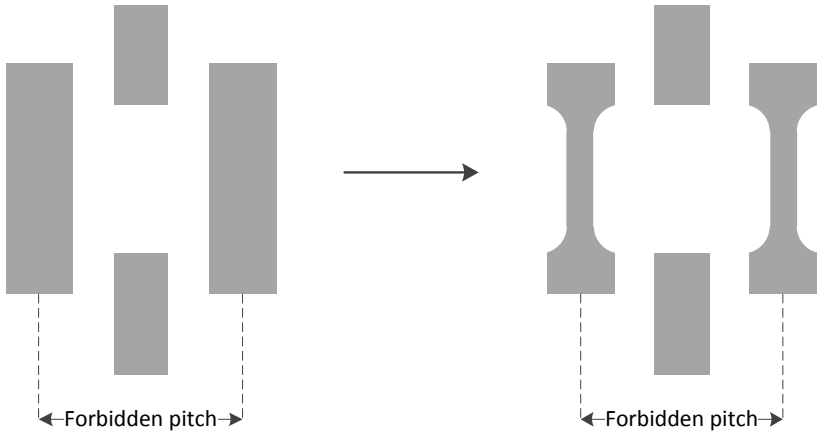


Figure 1.8: Example of the effect of forbidden pitches showing the intended layout (left) and the actual layout (right)

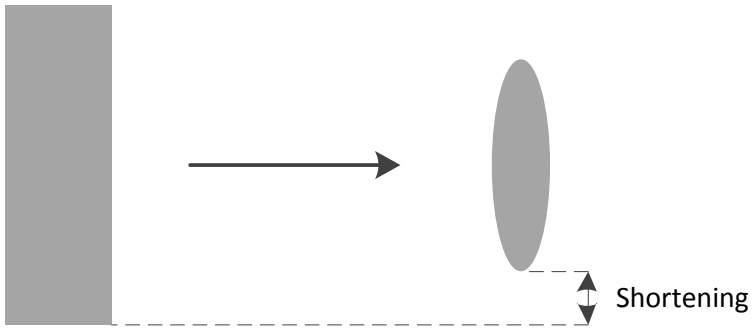


Figure 1.9: Example of line shortening showing the intended layout (left) and the actual layout (right)

Corner Rounding

This defect is when the square edges of wires in the layout become rounded. An example of this is shown in Figure 1.10. Figure 1.9 also shows the effect of corner rounding.

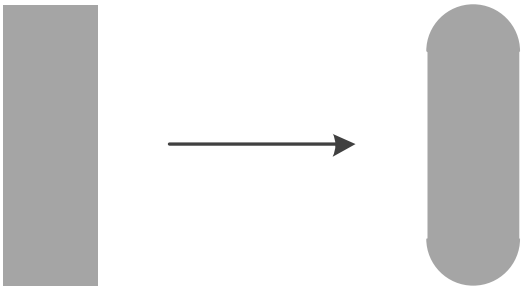


Figure 1.10: Example of corner rounding showing the intended layout (left) and the actual layout (right)

1.5 Faults

The detection of defects is important to manufacturers because faulty chips should, obviously, not be sold to the customers. The localization of defects is also important because it helps the manufacturer reduce the chance that these defects will occur by making changes to the layout or the design of the circuit. However, diagnosis is usually done at the logical level whereas the actual defects are on the physical level. This is due to the lower complexity and, hence, higher speed of logical simulations. Therefore, the defects must be abstracted and their effects represented at the logical level by *faults*, which become the targets for diagnosis. *Defect coverage* is a measure of how well a fault model represents the possible defects in a circuit, so the more defects the fault model can represent, the higher its defect coverage. There are several fault models which provide different ways of representing the effects of defects and in this section, the difference between these fault models will be explained.

1.5.1 The Different Fault Models

Stuck-At

A stuck-at fault occurs when a signal or one of its branches is stuck at 0 or 1, forming a stuck-at-0 or stuck-at-1 fault, respectively. Since each stuck-at fault affects only one node, it is classified as a single-net fault. This fault model has been proven to result in high defect coverage and can be used to model most of the defects possible during manufacture. However, there are still some defects which it cannot model [6].

As mentioned earlier, it has been proven that stuck-at fault coverage below one hundred percent means that there is a logic redundancy in the circuit [9]. Optimization of the circuit to remove the redundancy will therefore allow complete stuck-at fault coverage. However, in some cases, logic redundancy might be introduced on purpose.

Bridging

This model has several variants but all of them aim to represent the behaviour of a defect which connects two or more adjacent signals forming a bridge. Since each bridging fault affects more than one node, it is classified as a multi-net fault. This model can represent some defects which cannot be represented using the stuck-at fault model [6]. Hence, it should be used to complement the stuck-at fault model. The following is a list of some common bridging fault models.

Wired-AND

: In this variant, if one of the signals is a 0, then the others are also set to 0.

Wired-OR

: In this variant, if one of the signals is a 1, then the others are also set to 1.

Dominant-AND

: In this and the dominant-OR variants, there are aggressors and victims. The aggressors do not change whereas the victims can be affected by the aggressors. In this variant, if one of the aggressors is 0, then all of the victims are also set to 0.

Dominant-OR

In this variant, if one of the aggressors is 1, then all of the victims are also set to 1.

Dominant-Bridge

In this variant, there is only one aggressor. The rest of the signals are victims. The victims are all set to the value of the aggressor, whether it is 0 or 1.

1.6 The GDS format

A GDS file is used to describe the geometrical layout of the circuit using layers of material such as the ones used in CMOS technology. It contains the final layout which is used to fabricate the circuit.

1.6.1 Hierarchy

- Each GDS file is a Library,
- Each Library consists of several Structures (also known as cells).
- Each Structure has a name and consists of Elements.
- There are several types of Elements. The most common are the following:
  - Boundary
  - Path
  - SRef
  - Text

1.6.2 Elements

Boundary, Path and Text elements have a Layer number attribute. This number specifies the material layer to which the element belongs to. The number itself is arbitrary and must be tracked by the designer to map it to the appropriate fabrication layer. In addition, all elements have at least one

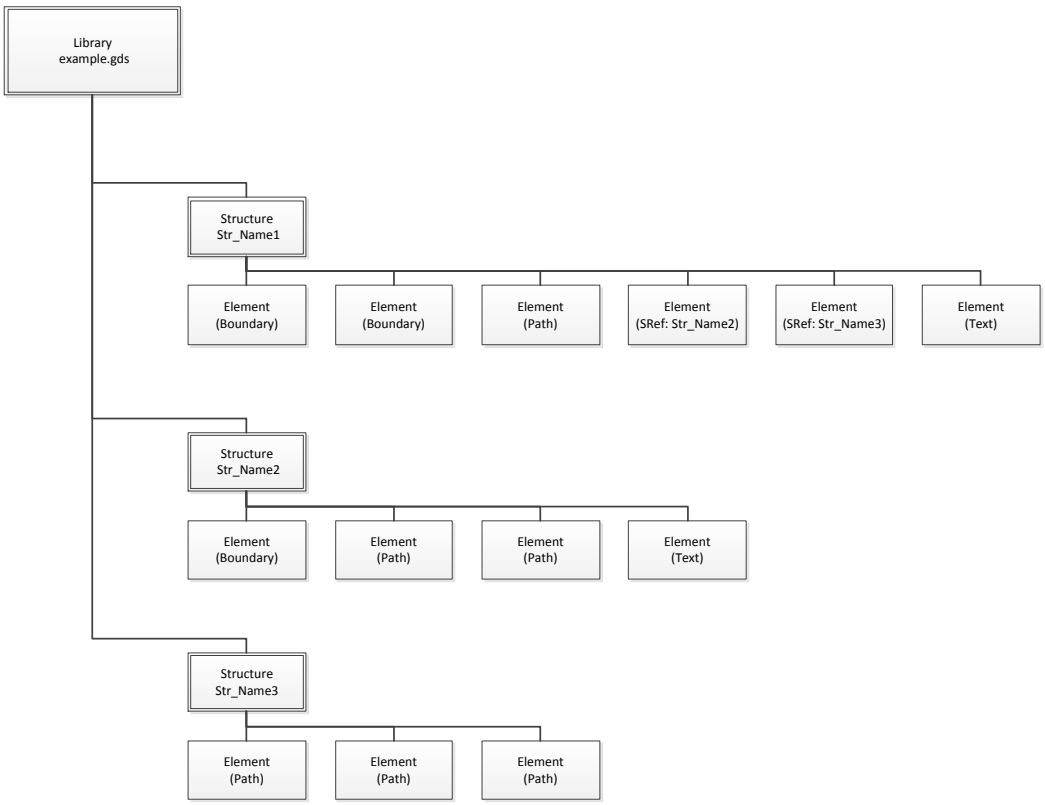


Figure 1.11: Example of a GDS library

Point. Points are simply pairs of XY-coordinates which allow positive and negative numbers. They are relative to other Points in the same Structure meaning that if every Point in the root Structure is shifted by +100 in the x-direction, the resulting layout will be equivalent. A difference will only be seen if the shift is performed to Points in a Structure which is referenced in another Structure. This is due to the way Structures are instantiated by SRefs as described below.

Boundary

Boundaries are simply polygons which have two attributes, a Layer number and a list of Points. The list of Points gives the corners of the polygon and the polygon is formed from those Points by joining every two adjacent Points in the list with a straight line. Note that in the GDS file, Boundaries must be closed explicitly, therefore the first Point in the list must have the same coordinates as the last Point. An example of a Boundary is shown in Figure 1.12.

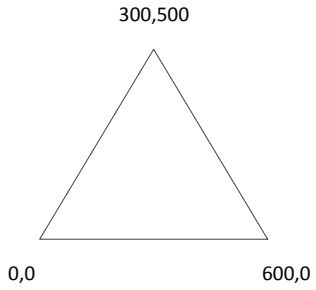


Figure 1.12: Example of a Boundary element

Path

A Path is a wire defined by its Layer number, its list of connecting Points, its width and how the shape of the wire looks like at the endpoints. It has a minimum of two Points, defining a Path made out of one rectangle. It also has a *Net* attribute to map it to the appropriate node in the logic circuit. The Pathtype attribute defines how the shape of the wire looks like at the endpoints. There are four Pathtypes available, the default being Pathtype 0, and they are explained below.

Pathtype 0 Paths with this Pathtype are square-ended and their ending-edges end at endpoints. An example of such a Path is shown in Figure 1.13(a).

Pathtype 1 Paths with this Pathtype are round-ended. An example of such a Path is shown in Figure 1.13(b).

Pathtype 2 Paths with this Pathtype are square-ended and their ending-edges are extended after the endpoints by half of the defined width. An example of such a Path is shown in Figure 1.13(c).

Pathtype 4 Paths with this Pathtype have ending-edge extensions with variable length.

(a) Example of Pathtype 0

(b) Example of Pathtype 1

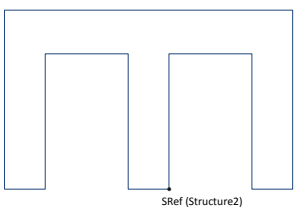
(c) Example of Pathtype 2

Figure 1.13: Examples of Pathtypes

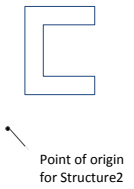
SRef (Structure Reference)

An SRef is used to instantiate another Structure at a defined Point and it can have optional attributes to specify mirroring or rotating the instantiated

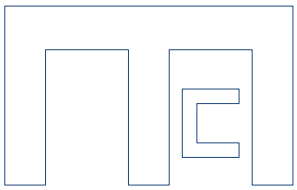
copy of the Structure. The Structure to be instantiated is referred to by name therefore the SRef must have an SNAME attribute to determine which Structure to instantiate. When instantiating a Structure, any Points directly in it (ie. not inside a Structure instantiated in it) are shifted by the SRef Point. This means that the SRef Point corresponds to the (0,0) point in the original definition of the instantiated Structure.



(a) Structure1 including SRef to Structure2



(b) Structure2



(c) Structure1 after flattening

Figure 1.14: SRef flattening example

SRefs allow a hierarchical format, with Structures containing Structures, which contain other Structures, etc. Flattening is the process of replacing references to Structures with the layout of the Structures themselves. An example of this is shown in Figure 1.14. Figures 1.14(a) and 1.14(b) show the two Structures before flattening and Figure 1.14(c) shows them after flattening. Notice that the only indication as to which Structure is the root Structure is that its the Structure which is not referenced in another one. However, this might not always be the case since the GDS Library might contain redundant Structures which have not been referenced. One case where this might happen is when merging a user-created GDS Library with

a Standard Cell Library in GDS format; the user-created Library will use some of the Structures defined in the Standard Cell Library but not all of them, causing the remaining to be redundant. Therefore, it is important for the designer to keep track of the name of the root Structure in order to avoid problems. Also note that an SRef Element does not have a Layer number. This is because the Structure to be instantiated is not confined to one layer and might have Elements in any layer, with each Element specifying its own Layer number.

Text

A Text element is simply a text label on the layout which has no effect on the fabrication process. It is defined by a Layer number to indicate the layer the text is relevant to, a string of characters, and a Point to which this string is attached on the layout.

Chapter 2

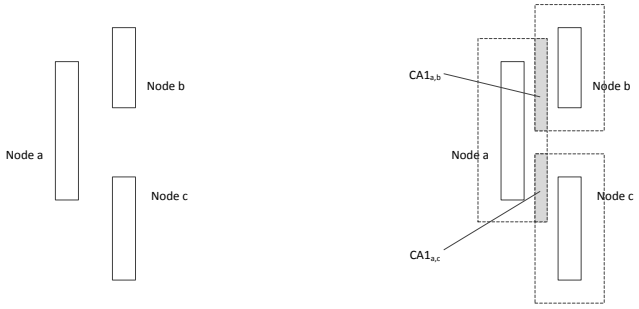
Inductive Fault Analysis

2.1 What is IFA?

As mentioned in the introduction, IFA is generally the process of extracting an ordered list of bridging faults by analyzing layout and defect data [5] [10]. In this thesis, the list is ordered in descending order of weighted critical area (WCA), which is a model for the probability that this fault will occur. In order to calculate the WCA, the critical area (CA) must first be obtained. The CA for a bridge, given a certain defect size, R, is defined as the area in which the centre of a defect with a radius equal to R can occur in order to cause the bridge. It is calculated as follows.

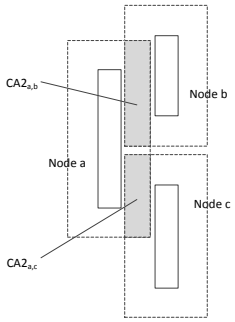
Assuming the layout of the circuit is a collection of rectangles belonging to several nodes, each one of those rectangles is expanded by the defect size, Ri, and then the overlapping area between them is obtained. This overlapping area is the CA and is calculated for each set of nodes. To calculate the WCA_{Ri} for a set of nodes, its CA is multiplied by the probability that a defect of size Ri will occur during manufacture [10]. This is repeated for every given defect size and for each set of nodes, the WCA_{Ri} values for it are added up to form WCA_{total}. An example of this calculation is shown in Figure 2.1, assuming that $R1$ and $R2$ are the only defect sizes given, having probability of occurrence $p1$ and $p2$ respectively. For the example:

$$WCA_{R1} = CA * p1$$
$$WCA_{R2} = CA * p2$$
$$WCA_{total} = WCA_{R1} + WCA_{R2}$$



(a) Original layout

(b) Layout after expansion by R1



(c) Layout after expansion by R2

Figure 2.1: Example of critical area calculation

2.2 The Chosen Algorithm for IFA

Before explaining the algorithm, a few assumptions must be made. First, the layout must be of Manhattan design, meaning that all the geometric shapes in the layout are rectilinear. Second, all the shapes are rectangles. We distinguish between these two assumptions, even though the second is a subset of the first, because the first is easy to achieve using placement and routing tools such as Cadence®Encounter. The second assumption, however, is dealt with separately and this is explained in Section 3.

The goal of the algorithm is to detect intersections between rectangles in the same layer, calculate the WCA formed by these intersections, and finally

produce an ordered list of multi-node bridging faults. To detect intersections, interval trees are used [11, 12, 13]. Each interval represents the y-coordinates of the corners of a rectangle. Interval trees are implemented as follows.

2.2.1 Interval Trees

Creating a static tree

First, a *static* binary tree is created. To clarify, assume that a set of intervals, S = ($\langle$1,4$\rangle$, $\langle$2,3$\rangle$, $\langle$2,5$\rangle$, $\langle$4,7$\rangle$, $\langle$5,7$\rangle$), exists. The leafs of this tree are the distinct endpoints of the intervals inside S. Each non-leaf node is tagged with a value equal to $\frac{n1+n2}{2}$, where $n1$ is the value of the rightmost leaf in the left child subtree and $n2$ is that of the leftmost leaf in the right child subtree. Therefore, the tree is created from the bottom up by first creating the leafs, then climbing up the tree and creating parents until the root is reached. Note that for a level of nodes within the tree with an odd number of nodes, the remaining single node will simply be promoted to the above level instead of creating a parent for it. This is because such a parent will have only child, which is that remaining node. Therefore, the extra node created will be useless and will only serve to adversely affect the performance. The static tree for set S in our example above is shown in Figure 2.2.

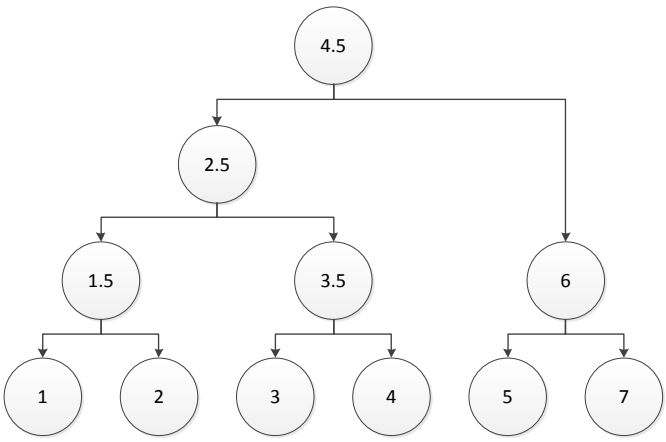


Figure 2.2: Example of a static tree

Inserting and deleting intervals

After the static tree is created, a *dynamic* tree is inserted on top of it. Each dynamic node represents an interval. In order to insert a dynamic node, the static tree is searched, starting with the root and going down, for a static node which is tagged with a value within the interval of the dynamic node. The first such static node which is found will have the dynamic node attached to it and will be called the fork node of that dynamic node. Also, any static node which is a fork node of a dynamic node is called an active node. Therefore, for our example of set S, the interval $\langle 1, 4 \rangle$, for example, would be attached to the active/static node (2.5). To delete a dynamic node, the node is simply detached from its fork node.

The detection of interval intersection is done after every insertion. After the insertion of an interval, A, three paths are defined on the static tree. P1 is the path from the root node to the fork node, P2 is the path from the fork node to the leftmost leaf below the fork node, and P3 is the path from the fork node to the rightmost leaf below the fork node. For our example interval $\langle 1, 4 \rangle$, P1 is $[(4.5) \rightarrow (2.5)]$, P2 is $[(2.5) \rightarrow (1.5) \rightarrow (1)]$, and P3 is $[(2.5) \rightarrow (3.5) \rightarrow (4)]$. The intervals which are flagged as intersecting with A are as follows.

- All other intervals attached to the fork node intersect with A.
- Each active node in P1 is checked to see if its tagged value is to the right or left of A.
  - If it is to the right, each interval, B, attached to the active node is checked using the following. If the left end of B is less than the right end of A, then A and B intersect.
  - If it is to the left, the case is analogous. If the left end of A is less than the right end of B, then A and B intersect.
- Each active node in P2 and P3 is checked to see if its tagged value lies within A.
  - If it does, then all intervals attached to it intersect with A.
  - If it does not, then a check equivalent to the one for active nodes in P1 is performed.

2.2.2 Application of Interval Trees

In order to detect rectangle intersections, the above algorithm is used as follows. The set S contains intervals which are the y-coordinates of the left and right edges of all the rectangles in the given layer. For example, if the rectangles are as shown in Figure 2.3, S becomes equal to ($\langle$1,5$\rangle$, $\langle$3,7$\rangle$, $\langle$4,8$\rangle$) and the corresponding static tree becomes as shown in Figure 2.4. This means that the leafs of the static tree are the distinct y-coordinates of the rectangles. For the dynamic tree, the order of insertion and deletion of the intervals depends on the x-coordinate of each interval. The left edge of a rectangle causes an insertion and the right edge causes a deletion. Between the insertion and deletion of this interval from the dynamic tree, the rectangle represented by this interval "exists" in the tree. If another interval is inserted before the first one is deleted and it intersects with the first interval, as determined by the checks above, then this means that the two rectangles represented by these two intervals overlap. The region of overlap, called a critical area rectangle (CAR), is from the x-coordinate of the second interval to the x-coordinate of the interval which is deleted first, and from the y-coordinate of the highest bottom edge to the y-coordinate of the lowest top edge.

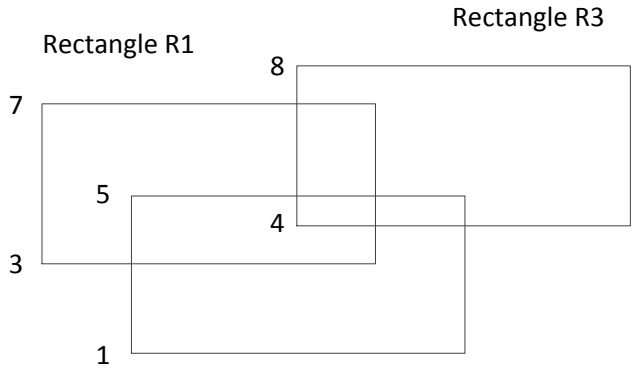


Figure 2.3: Example of overlapping rectangles

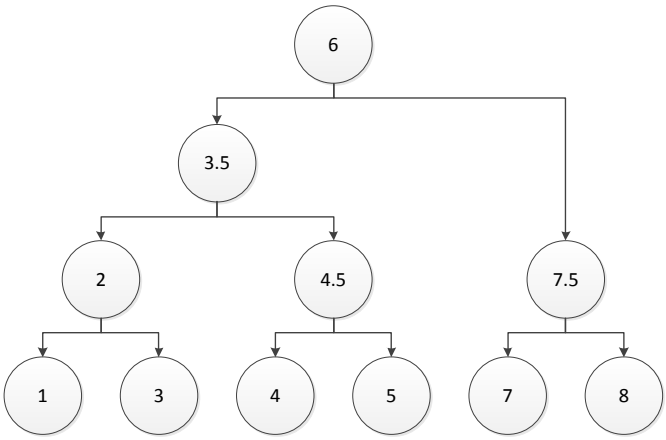


Figure 2.4: Static tree for Figure 3.2

2.2.3 Merging CARs

After obtaining CARs from rectangle intersections, the CARs must be merged to obtain CAs. This step of the algorithm is where multi-node bridges are obtained. Any overlap between CARs forms a CA tagged with all of the nodes associated with the overlapping CARs. An example of this is shown in Figure 2.5. The algorithm for this stage has two loops, one nested inside the other, and works as follows. The outer loop iterates over the distinct x-coordinates of all the CARs and divides the diagram into columns. The inner loop iterates over the distinct y-coordinates of the edges of all the CARs lying on the current x line. At each y-coordinate, the corresponding edge is checked to see if its a top or bottom edge. If it is a top edge then the nets of the CAR are added to a set S. If it is a bottom edge then the nets are removed from set S. Therefore, set S keeps track of the nets of the areas between the y-coordinates. At each y-coordinate, a WCA is also created unless the above area in the column is empty or both nets introduced at this y-coordinate already exist in S. The obtained CAs are then weighted and the WCAs, along with their associated nodes, are added to the fault list.

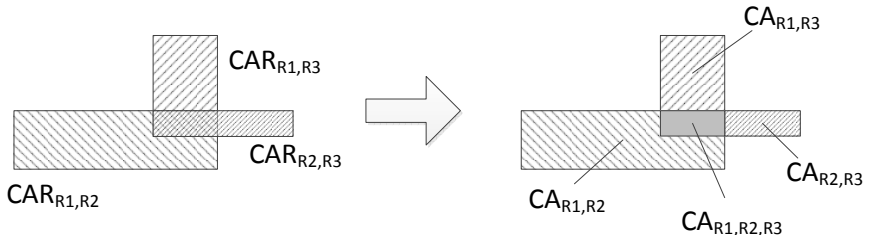


Figure 2.5: Example of CAR merging

2.3 Previous Work

There a few algorithmic differences between the chosen algorithm and CARAFE. The first is that CARAFE uses a line sweeping algorithm to detect rectangle intersections whereas the chosen algorithm uses interval trees. Another difference is that in CARAFE, rectangle intersections are searched for and detected for every defect size in the given defect data, whereas in the chosen algorithm, rectangle intersections are performed only once for the maximum defect size and then the CARs for the other defect sizes are calculated from the maximum CARs. This avoids unnecessary computations and hence results in lower execution times.[4]

The algorithm used by FedEx for fault extraction and circuit extraction is a scanline algorithm [14]. It creates a scanline which moves from top to bottom. Processing along the scanline occurs in the x-direction. However, as mentioned before, this algorithm only extracts two-node bridges. Furthermore, FedEx trades off accuracy for lower execution time since it only approximates WCAs. Similar to the chosen algorithm, LOBS only scans the layout once and then computes WCAs for all defect sizes. LOBS also uses a sliding window in order to reduce memory usage [15]. However, the algorithm used by LOBS also supports only two-node bridges.

Chapter 3

Implementation

The programming language used in the implementation is the Perl [16] programming language. It was chosen due to its excellent text manipulation capabilities, the core of which is the matching of regular expressions, and also due to the fact that it can be object-oriented.

3.1 Preparation

In order to implement the algorithm described in Section 2.2, the input for it must first be prepared. The steps for this preparation are explained in this section.

3.1.1 Obtaining the GDS file

At the very beginning, the circuit to extract bridging faults for is available to us in VHDL or Verilog form. In order to obtain the GDS file for the circuit, the circuit must first be placed and routed to obtain a physical layout for it. This placement and routing is done using the Cadence®Encounter tool [17]. The tool places instances of standard cells from the standard cell library in appropriate positions and then interconnects the inputs and outputs of these cells in order to obtain the circuit's function. The process of placing the standard cells is the placement stage, whereas the process of interconnecting the cells is called the routing stage. As mentioned before, the layout produced here must be of Manhattan design. The standard cell library used in this thesis is the NanGate 45nm Open Cell Library [18].

After a physical layout is automatically generated by Encounter, it is exported as a GDS file with the option to add net names to Paths as an attribute with attribute number 100. The number 100 here is arbitrary but it is the one which was chosen for this implementation. It is important to note that the GDS file obtained here is a binary file. In order to make the parsing stage easier and allow manual inspection of the GDS file, the binary GDS file is converted to an ASCII text file containing a clearer description of the layout. This conversion is performed using the OwlVision tool [19]. Therefore, the output at this point is a plain text file containing the GDS library.

3.1.2 Parsing

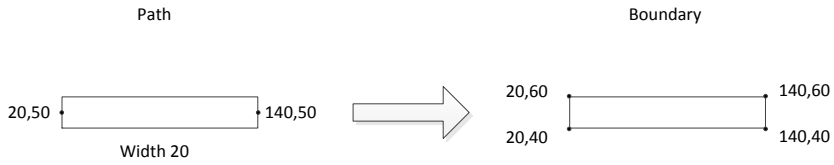
After obtaining the text file, the GDS library described in it must be extracted and parsed into an object-oriented internal data structure. The internal data structure was designed to mimic the structure of the GDS library. Therefore, it contains Structure, Boundary, Path, SRef and Point objects equivalent to their counterparts in the GDS library and has the same hierarchy shown in Figure 1.11. Note that the net names of the Paths provided by Encounter must be attached to their corresponding Path objects in order for them to be used in bridge extraction. Also, Paths of Pathtypes 1 and 4 as shown in Section 1.6.2 are not supported since all of the Paths produced by Encounter's placement and routing are either Pathtype 0 or 2.

After this stage, the state of the internal data structure is stored on disk. This is to avoid having to parse the text file every time a modification to the defect data is needed. For subsequent runs, the stored state can simply be used. This allows parsing to be done only once for each circuit.

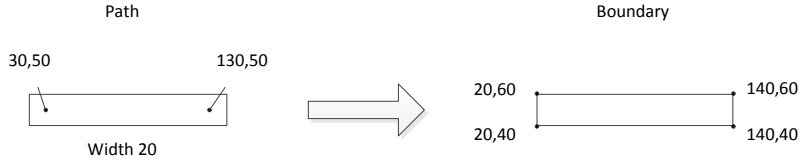
3.1.3 Replacing Paths with Boundaries

After parsing, Paths are converted to Boundaries. This is done to allow easier manipulation of the Paths since Paths are defined by their endpoints whereas Boundaries are defined by their corners. This step supports only two-Point Paths because Encounter only produces two-Point Paths therefore there was no need to support Paths with more than two Points. However, an extension to support multi-Point Paths is not difficult to implement if it is needed. Another part of this step is attaching to the new Boundary the net name of the Path provided by Encounter.

The way each of the corners of the rectangle is calculated depends on the Pathtype and the orientation of the Path. Since the layout is of Manhattan design, the Path can have Pathtypes 0 or 2 and can be either horizontal or vertical. This means that only four cases, formed by combinations of the different Pathtypes and orientations, have to be considered. Examples are given in Figures 3.1(a) and 3.1(b), for Pathtypes 0 and 2, respectively, when the Path is horizontal. Notice that any difference in x- or y-coordinates between the Paths' endpoints and the corners of the Boundaries is half the width of the Path. A similar calculation occurs for vertical Paths.



(a) Conversion for Pathtype 0



(b) Conversion for Pathtype 2

Figure 3.1: Examples of Path to Boundary conversion

3.1.4 Splitting Boundaries

The next stage in the preparation process is the splitting of Boundaries into rectangles. This is an optional stage because the Paths which are converted to Boundaries are all two-Point Paths, and hence already produce rectangles.

Furthermore, any original Boundaries which were not previously Paths, and hence have no nets attached to them, are disregarded as described in Section 3.1.5. However, future extensions of this work might create support for multi-Point Paths or a way to add net names to the original Boundaries. Depending on the implementations of these extensions, they might require the splitting of non-rectangular Boundaries into rectangles and, therefore, this stage was included.

For this stage, a new algorithm was created to implement the splitting of Boundaries. This algorithm assumes that there are no holes in the Boundaries. For example, a Boundary such as the one shown in Figure 3.2 is invalid. In the following section, the algorithm will be explained and examples will be given to clarify. A flow chart showing the steps of the algorithm is provided in Figure 3.4.

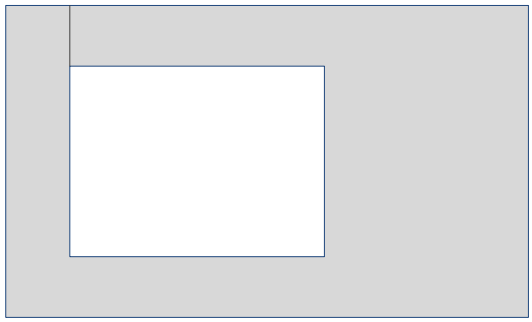


Figure 3.2: Example of a Boundary with holes

Detecting a hill

The first step in the algorithm is detecting a "hill". This hill is simply a sequence of four Points in the Boundary's Points list which signifies the protrusion of a rectangle at the top of the Boundary. The Points array is first made clockwise, if it is not already, and is then traversed until four Points matching the pattern for the hill are detected. This pattern is [y2>y1, x3>x2 and y4<y3], where the numbers signify the order of each Point in the sequence.

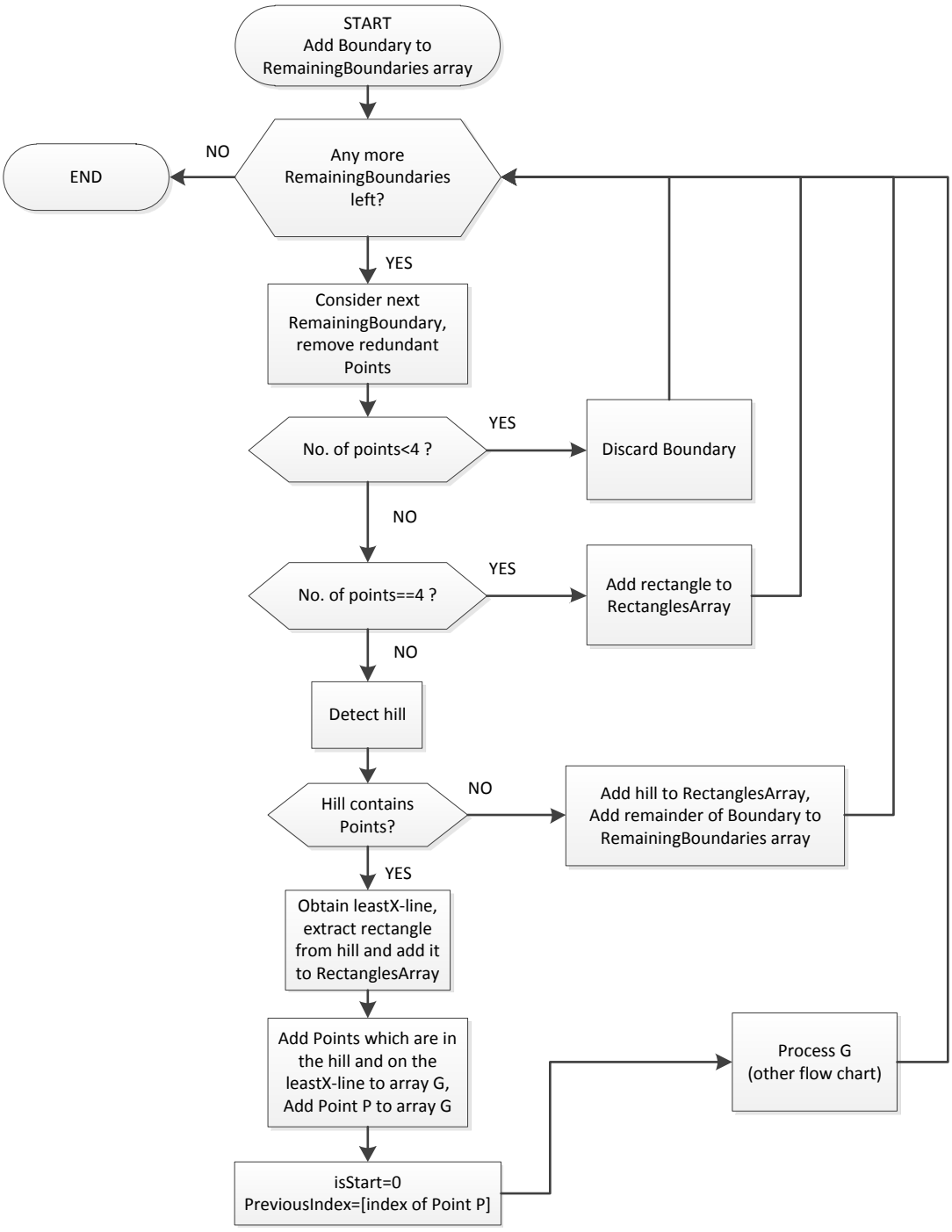


Figure 3.3: Flow chart for Boundary splitting

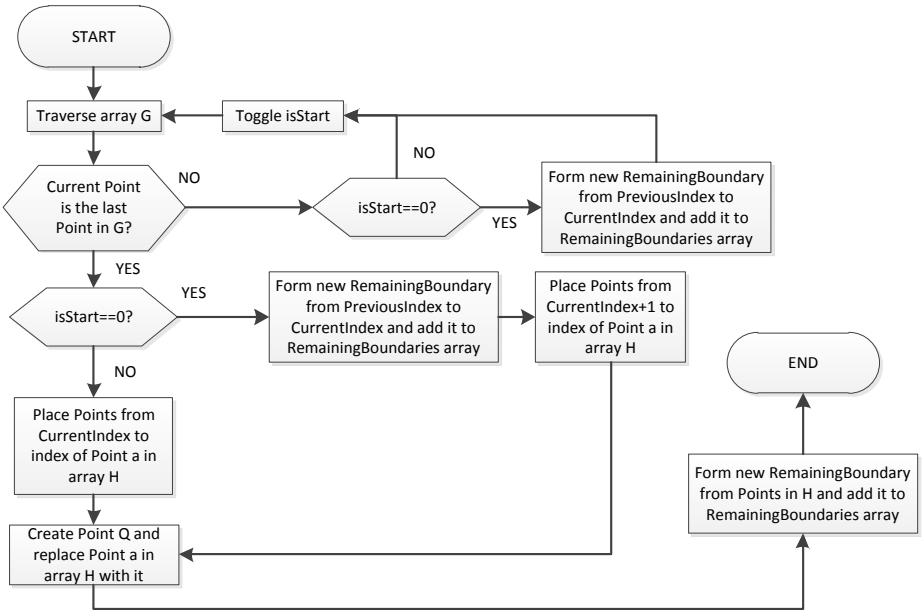


Figure 3.4: Flow chart for processing G

Notice how there is no need to check for [x1==x2, y2==y3, x3==x4]. This is because the Boundaries are rectilinear and have no redundant points, so if two consecutive Points differ in their y-coordinates, they will have the same x-coordinate, and vice versa.

As an example, assume that the Boundary to be split is the one shown in Figure 3.5. The hill is the shaded area and the four Points which form the sequence are indicated in the figure as A, B, C and D. Note that the lower edge of the hill has the y-coordinate of Points A or D, whichever is higher. After this, the corners of the hill become labeled as a, b, c and d and if a or d does not exist in the Boundary's Points array, it is added to it, temporarily creating a redundant Point. There are two cases to consider after detecting a hill: the hill contains Points, or it does not. Therefore, the next step after detecting a hill is checking to see whether it contains Points.

Figure 3.5: Example of a Boundary which has no Points inside the hill

Hill does not contain Points:

If the hill does not contain Points, such as in the Figure 3.5, then it is cut off and forms a rectangle which is added to the RectanglesArray. This array keeps track of the rectangles formed by splitting the Boundary and, in the end, will contain all the rectangles obtained from the Boundary. The remainder of the Boundary (the unshaded area in the figure) is added to the RemainingBoundariesArray. As the name suggests, this array keeps track of

the remaining parts of the Boundary which have not yet been processed and might not be rectangles.

Hill contains Points:

Extracting a rectangle

The second case, which is when the hill does contain Points, is more complicated. The hill cannot be simply cut off and made into a rectangle as this will give incorrect results. A rectangle must first be extracted from it and added to the RectanglesArray and after that, the remaining parts of the hill, along with the remaining part of the Boundary below the hill, must be added to the RemainingBoundariesArray for further processing as they also might not be rectangles. The rectangle which is extracted from the hill is the biggest rectangle possible formed from the left edge of the hill. The right edge of this rectangle is found by finding the Point inside the hill with the least x-coordinate. This x-coordinate is the x-coordinate of the right edge of the rectangle. The top and bottom y-coordinates of the rectangle are the y-coordinates of the top and bottom edges of the hill, respectively. An example of this case is shown in Figure 3.6. The extracted rectangle is shaded in grey and the area of the Boundary inside the hill, which includes the rectangle, is patterned with diagonal lines.

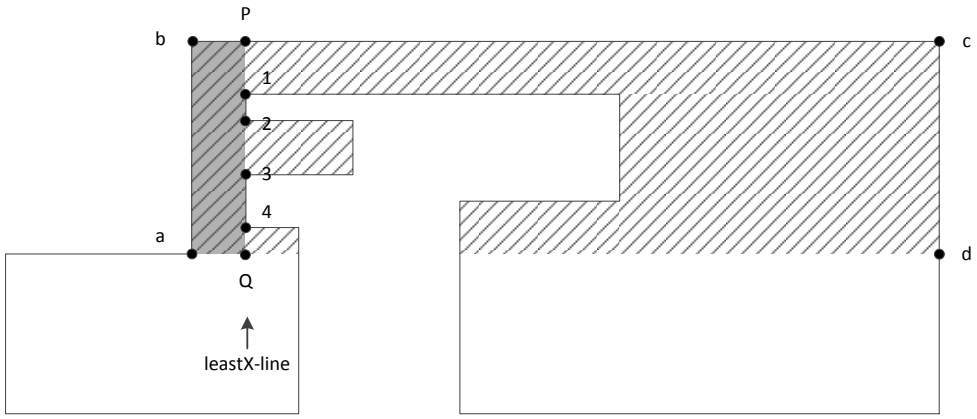


Figure 3.6: Example of a Boundary with Points inside the hill

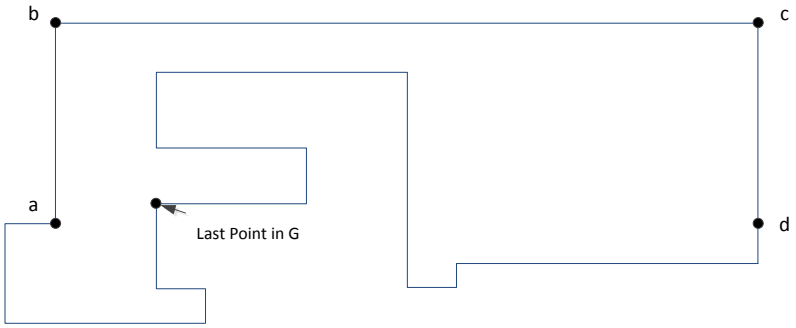
Obtaining Points on leastX-line

The remainder of the hill is obtained as follows. First, the Points on the leastX-line inside the hill are searched for in the Boundary's Points array and copied to another array, G, where they are sorted in descending order of their y-coordinates. Then, Point P, as shown in Figure 3.6, is created and added to the Boundary's Points array at the appropriate position (i.e. between Point b and Point c). Note that Point P can never exist in the Boundary's original Points array because if it did, the location of the hill would be different. The method which returns all the Points on the leastX-line also returns their indices in the Boundary's Points array to aid in the next step.

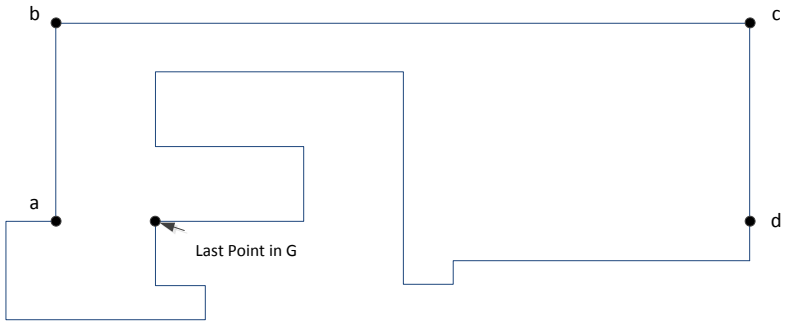
Processing G

Before traversing array G, a flag called isStart is set to 0. This flag indicates whether the next Point in G is the beginning or the end of a new Boundary to be cut off. Also, a variable called PreviousIndex is set to the index of Point P in the Boundary's Points array. Array G is then traversed and for every Point, its index in the main Boundary's Points array is stored in a variable called CurrentIndex. For every Point in G except the last one, if isStart=1, then isStart is simply toggled and changed to 0 and the Point's index in the main Boundary's Points array is stored in PreviousIndex. If isStart=0, however, then a new Boundary is formed from the Points in the main Boundary's Points array, starting from PreviousIndex to CurrentIndex. This new Boundary is then added to the RemainingBoundariesArray, the isStart flag is toggled and the next iteration of the loop is started. It is important to note that this part treats the main Boundary's Points array as circular, meaning that if the end of the array is reached before CurrentIndex (or, in other words, [CurrentIndex<PreviousIndex]), the counting continues from the beginning until CurrentIndex. The remaining Boundaries inside the hill which are added to the RemainingBoundariesArray for our example in Figure 3.6 are (P to 1) and (2 to 3).

When the last Point in G is reached, again two cases can occur. The first case is when isStart=0. This case indicates that the number of Points in the G array is odd and further subdivides into two subcases: when the last Point lies above the lower edge of the hill, and when it lies on it. These two subcases are shown in Figure 3.7 and, for-



(a) Last Point above the lower edge of the hill

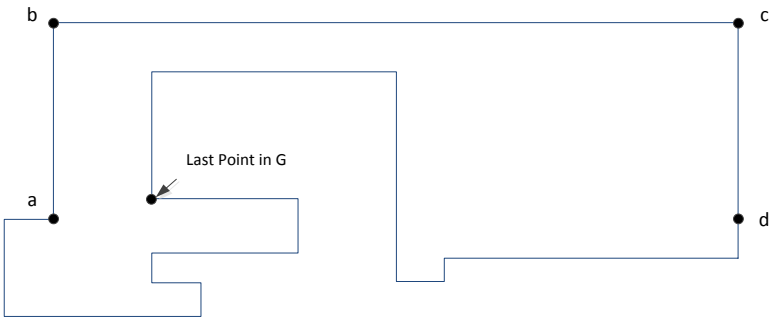


(b) Last Point on the lower edge of the hill

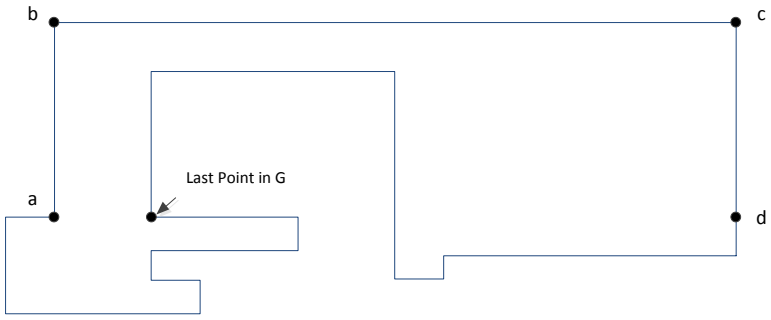
Figure 3.7: Examples of when isStart=0 for the last Point in G

tunately, the solution for them is identical. The Points in the main Boundary's Points array from PreviousIndex to CurrentIndex are first used to form a new Boundary which is added to the RemainingBoundariesArray. Then the Points in the main Boundary's Points array from CurrentIndex+1 to the index of Point a are placed in a new array H.

The second case is when isStart=1. This case indicates that the number of Points in the G array is even and also further subdivides into two subcases which are the same as for the previous case. These two subcases are shown in Figure 3.8 and, again, their solutions are identical. In this case, the Points in the main Boundary's Points array from CurrentIndex to the index of Point a are placed in array H.



(a) Last Point above the lower edge of the hill



(b) Last Point on the lower edge of the hill

Figure 3.8: Examples of when isStart=1 for the last Point in G

In both cases, a Point, Q, which is the bottom right corner of the extracted rectangle, is created and replaces Point a at the end of array H. The last remaining Boundary to be added to the RemainingBoundariesArray is then formed from the Points in H. For our example in Figure 3.6, that last Boundary is (4 to Q).

Note that there are cases where that last remaining Boundary does not actually exist, as shown in Figure 3.9, and which will cause H to contain only two Points with the same coordinates. Therefore, it is necessary that these 2-Point Boundaries are removed from the RemainingBoundariesArray. Another case which must be taken care of is the existence

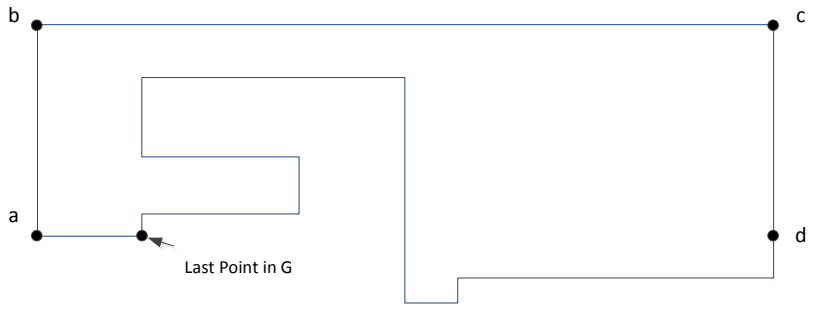


Figure 3.9: Example of a case where there is no last Boundary

of redundant Points in the remaining Boundaries' Points arrays. This occurs for some last remaining Boundaries as shown in Figure 3.10.

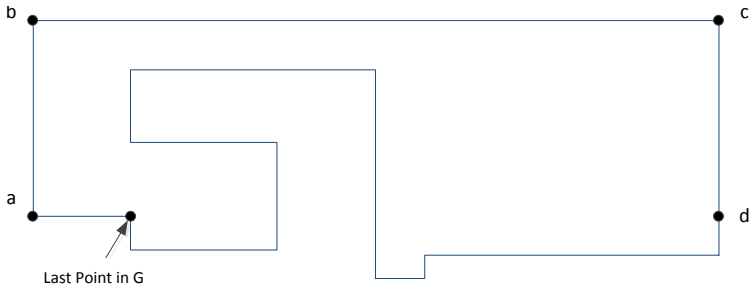


Figure 3.10: Example of a case which results in redundant Points

At the end, the entire process is executed again on each Boundary in the RemainingBoundariesArray, treating each one as a main Boundary. This can be implemented as a recursive algorithm, the end result of which is a set of rectangular Boundaries formed from the original Boundary and contained in the global RectanglesArray. The breaking condition of the recursion is when the recursive method is given a rectangular Boundary. In this case, the Boundary is simply added to the RectanglesArray and the method returns.

3.1.5 Preparing List of Boundaries

In this section, the list of rectangles which will be passed to the bridge extraction algorithm is filtered to remove netless Boundaries. Net names

are only added to Boundaries when they are created from Paths and since there are some Boundaries which were not previously Paths, there will exist some Boundaries which have no net name attributed to them. Since the bridge extraction algorithm relies on the rectangles' having net names, netless Boundaries must be removed. If they are not, bridges might be extracted involving unidentified nodes.

The effect of this removal had to be analyzed in order to ensure that the information obtained from the extraction was not misleading. This was done by manually comparing the converted GDS file to the visual layout produced by Encounter. For several circuits, the netless Boundaries were found to be of no significant effect on the the bridges extracted. This was because they were found in the outer areas of the chip and not in the areas containing the standard cells, and because their size was very small compared to the rest of the layout. Hence, it was decided that they will be ignored.

3.2 Extraction of Bridges

After the preparation stage is completed, the next step is to execute the algorithm presented in [4]. The inputs of the function which executes the algorithm are the array of rectangles after filtering, a list containing the layers to be considered in the algorithm, and a set of defect data. The set of defect data is a hash with the format

$$\{L1,\text{(defect data for layer 1)},\ldots,Ln,\text{(defect data for layer n)}\}$$

where (defect data for layer i) is

$$[\{\text{“}R\text{”},Ri_1,\text{“}P\text{”},Pi_1\},\ldots,\{\text{“}R\text{”},Ri_m,\text{“}P\text{”},Pi_m\}]$$

Ri_j is the radius of the j^{th} defect size for the i^{th} layer.

First, the rectangles are separated into different arrays according to their layer. Then, for each layer, the following occurs. All the rectangles are first expanded by the maximum defect size for that layer. Then, rectangle intersection is performed as described in Section 2.2. First, a static tree is created using the distinct y-coordinates of the corners of the rectangles as leafs. This is done using a recursive function which creates each level of the trees and then climbs up to the next level until the root node is reached. After the static tree is created, a table of events (insertions and deletions) is created to

signify the left (insertion) and right (deletion) edges of the rectangles, where each event is associated with a certain interval (the pair of y-coordinates of the rectangle) and the x-coordinate of the edge it corresponds to. The table of events is sorted in ascending order of the x-coordinate, with insertions preceding deletions for cases where there are several events with the same x-coordinate. To clarify, an example of the rectangles to be processed is shown in Figure 3.2. Table 3.1 shows the events table produced for these rectangles. Note that each Interval object is given two x-coordinates to mark when the interval will be added to the dynamic tree and when it will be removed. These x-coordinates are in addition to the y-coordinates which signify the actual upper and lower bound of the interval.

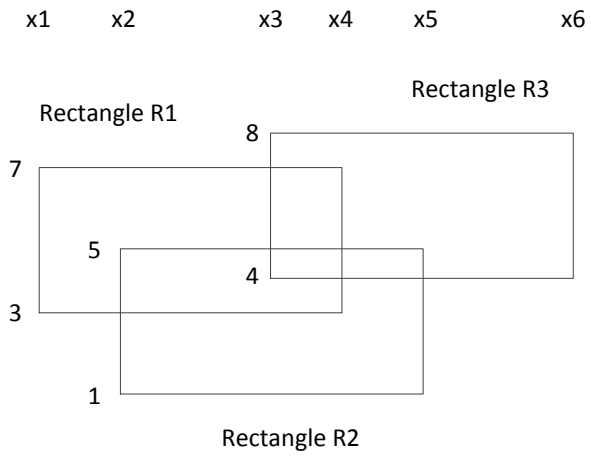


Figure 3.11: Example of overlapping rectangles to be processed

Event	I	I	I	D	D	D
Name	R1	R2	R3	R1	R2	R3
X	x1	x2	x3	x4	x5	x6
Interval	3,7	1,5	4,8	3,7	1,5	4,8

Table 3.1: Events table

After obtaining the table of events, the events are performed in order and the dynamic tree is produced on top of the static tree. Note how the x-axis is treated as the time axis, with events occurring in order of their x-coordinate. Intersections between rectangles are kept track of using an IntersectionsArray for each Interval object. This array holds the list of other intervals which are currently intersecting with the array's interval. For every insertion, if an intersection is detected between two intervals using the process described in Section 2.2, then they are added to each other's IntersectionsArray. For every deletion, if the deleted interval contains intervals in its IntersectionsArray (i.e. it is currently intersecting with other intervals), then the CAR produced with each of these intervals is calculated and added to the CARs array. Remembering that each Interval object contains variables to hold its insertion-x and deletion-x, a CAR is calculated by taking the highest insertion-x, the lowest deletion-x, the highest lower-y and the lowest upper-y. Table 3.2 shows the progression of the IntersectionsArray of each interval and the produced CARs for each deletion as the events table is traversed.

	IntersectionsArray after each event					
Interval	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**
R1	-	R2	R2, R3	#	#	#
R2	-	R1	R1, R3	R3	#	#
R3	-	-	R1, R2	R2	-	#
CARs created	-	-	-	$CAR_{R1,R2}$, $CAR_{R1,R3}$	$CAR_{R2,R3}$	-

Table 3.2: Progression of the IntersectionsArray of each Interval

At this stage, the CARs produced are the ones for the maximum defect size. For every defect size including the maximum, the CARs are shrunk by $\Delta R = R_{max} - R_{current}$ and then four six-tuples are created per CAR. The four six-tuples are (x1, y1, I, B, N1, N2), (x1, y2, I, T, N1, N2), (x2, y1, D, B, N1, N2) and (x2, y2, D, T, N1, N2), where N1 and N2 are the names of the nodes related to the given CAR, and $\langle$x1,y1$\rangle$ and $\langle$x2,y2$\rangle$ are the coordinates of the lower left and upper right corners respectively. These six-tuples are added to the list, L, which in the end will contain all the six-tuples for all the CARs for all the defect sizes. List L is then sorted in ascending order of x1 in the six-tuples. Note that the procedure of shrinking CARs allows the expensive process of interval intersection to be performed only once instead

of for each defect size.

After obtaining list L, the algorithm described by the flow chart in Figure 3.12 is performed. It processes L, maintains a list M and produces the fault list in an internal format. The main function of this algorithm is the merging of critical areas. The flow chart for the processing of M is shown in Figure 3.13. Note that the internal format of the fault list for each fault is

$$WCA, (\text{list of bridged nodes})$$

3.3 Processing the Internal Fault List

At this stage, we have a fault list containing WCAs for multi-node bridges. However, this list is still not ready for output for a few reasons. The first is that it can contain multiple entries for the same set of nodes. The WCAs for these entries have to be added and the entries merged into one entry in the fault list. The second reason is that it can contain faults which have a WCA of zero. This results when the rectangles are just touching but not overlapping after expansion resulting in zero critical area. An example of this case is shown in Figure 3.14. For a bridge to occur in this case, the centre of the defect must be exactly in the middle between the two rectangles in the layout. The third reason is that some faults will have only one net name. This occurs when overlaps occur between rectangles having the same net name. The fourth and final reason is simply that the fault list is not yet ordered. Therefore, the fault list is processed to merge multiple entries, filter out zero WCA entries, and then sort the list in descending order of WCA. This gives us the final fault list in the internal format, ready to be converted to ADAMA format.

3.4 Converting from Internal to ADAMA Format

In this step, the internal fault list is converted to a fault list in ADAMA format. The ADAMA format which is used is the versatile MultiLineFlip format shown below.

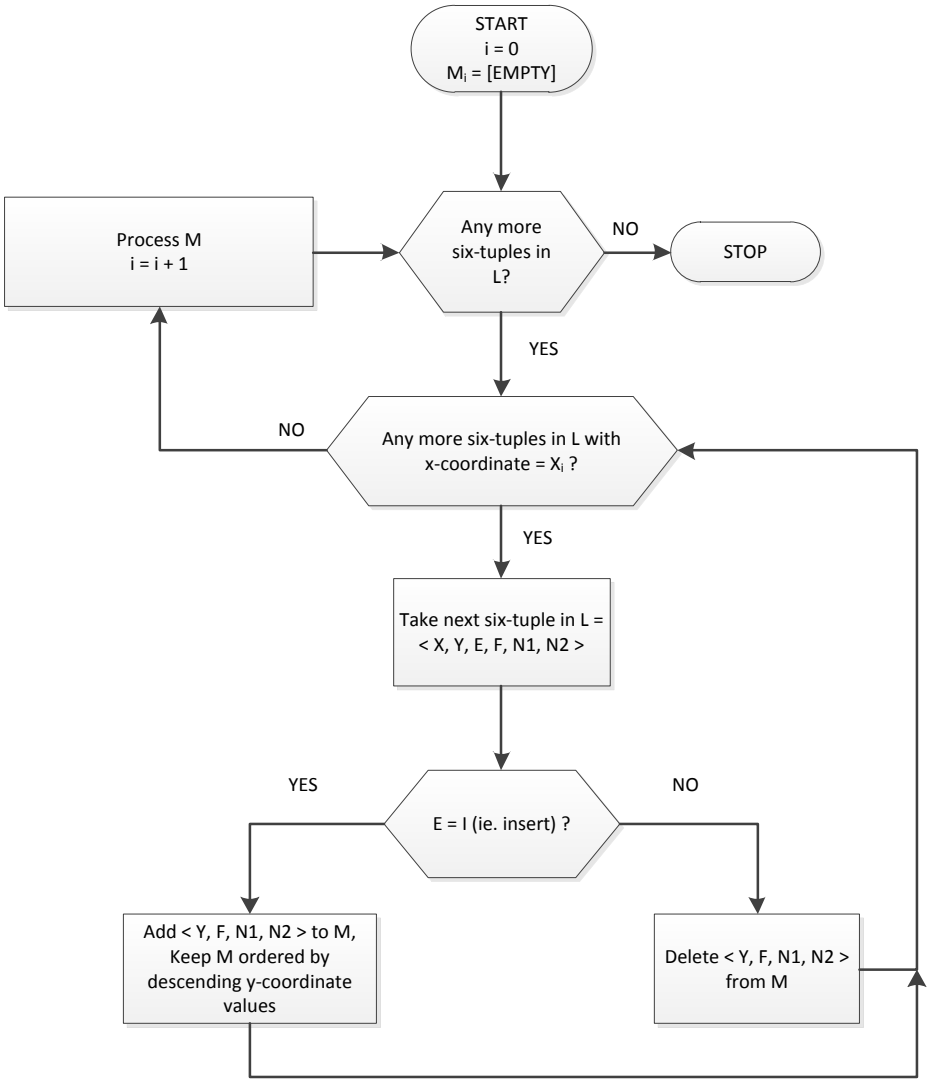


Figure 3.12: Flow chart to process L

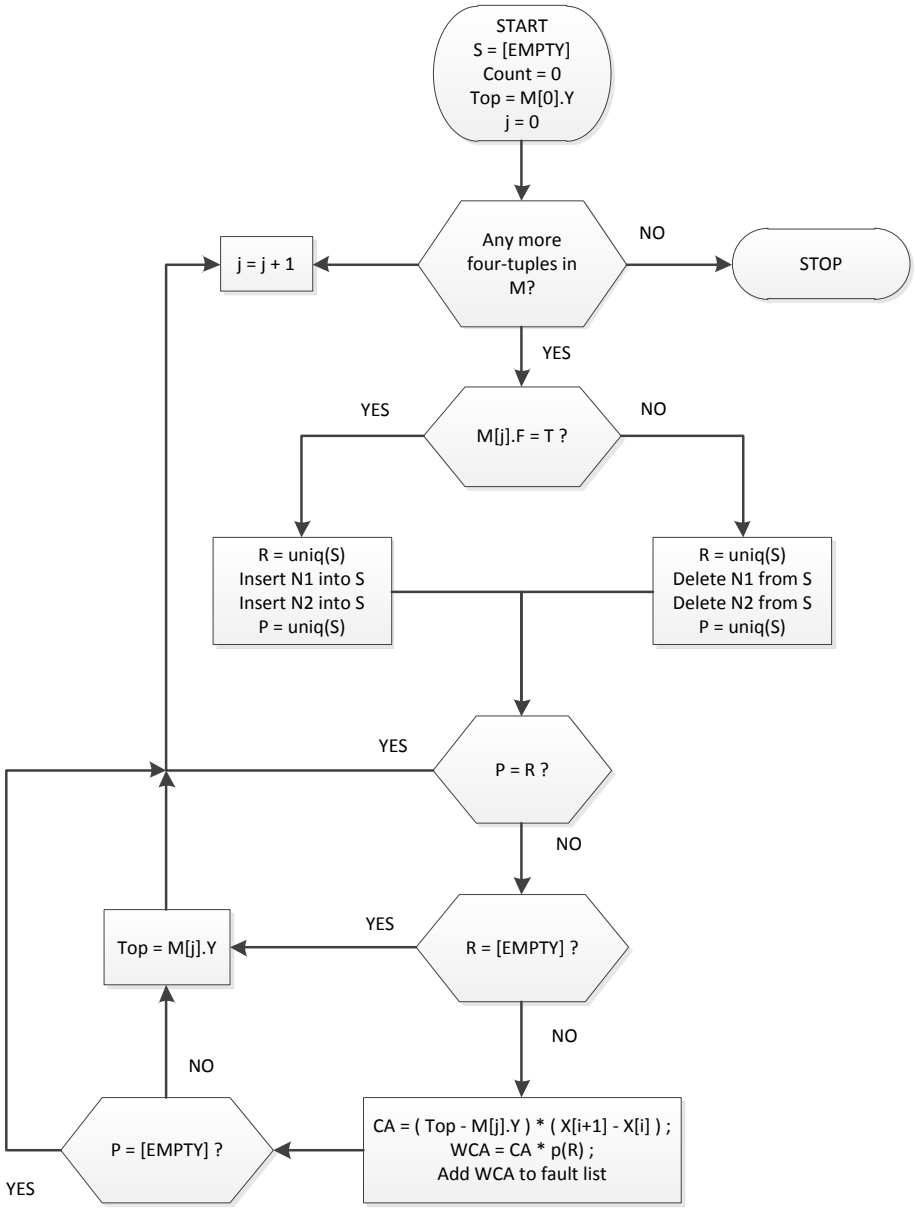


Figure 3.13: Flow chart to process M

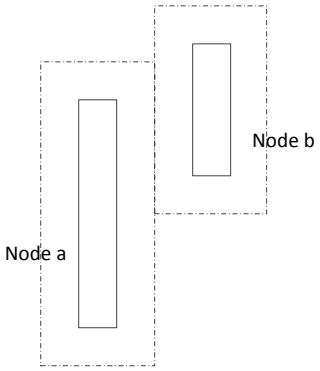


Figure 3.14: Example of a case causing a WCA of zero

MultiLineFlip { [# of aggressors] [list of aggressors] [# of victims] [list of victims] [fault model truth table] }

For the three dominant bridging fault model variants, only one aggressor is chosen for each MultiLineFlip due to limitations within ADAMA; the rest of the nets involved in the bridging fault are set as victims. For each multi-node fault in the internal fault list with n nets, n combinations are created, each time using a different net as the aggressor.

For the wired-AND and wired-OR variants, all the nets are set as victims. Therefore, only one MultiLineFlip is created for each fault in the internal fault list.

It is important to note that the nets in the internal fault list are named differently than they are in ADAMA. The net names in the internal fault list are identical to the net names produced by Cadence®Encounter, whereas ADAMA names nets according to their connections to the cell instances. Therefore, a necessary operation is mapping between the net names in the internal fault list and in ADAMA. The way this is done is as follows.

During the place and route stage, Encounter is used to output a VHDL file, in addition to the GDS file, containing the names of the instances and nets as they are named within Encounter (and subsequently within the GDS file and the internal fault list). This VHDL file is the one given to ADAMA as input, after a few modifications, such as removing 'inout' signals, are applied to it. These modifications are necessary in order to allow ADAMA to read the file in successfully and do not alter the operation of the circuit.

After ADAMA reads in the VHDL file, it names nets according to their connections to the cell instances and disregards most of the net names used by Encounter. Therefore, the VHDL file is read in by our implementation and the disregarded net names are mapped to their names according to ADAMA.

A special case occurs with complex cells. This is because ADAMA splits up these complex cells into simple gates and names even the outer signals of the complex cell blocks according to the way each complex cell is split up. To overcome this problem, a library was created which contains every complex cell used. This library defines the outer signals of the complex cells appropriately (i.e. according to ADAMA) and maps them to the correct net names in the internal fault list. Fortunately, only a few entries were required in this library and, naturally, they all corresponded to complex cells within the NanGate 45nm Open Cell Library.

Chapter 4

Results

4.1 Bridge Extraction

The implementation was used to analyze and perform IFA on four ITC'99 circuits: B02, B06, B20 and B22_1. Defect data was adjusted for each circuit to take into account the differences in dimensions. Table 4.1 shows the information obtained after analysis.

4.2 Simulation and diagnosis

For comparison, ADAMA was used to generate n random faults where n is the number of faults of that fault model obtained from IFA. After obtaining the faults lists (IFA and random) for each bridging fault model, ADAMA was used to inject and simulate the faults and perform diagnosis. The results of the diagnosis are shown in Table 4.2.

When considering each fault model separately, the fault coverages for circuits B06, B20 and B22_1 for IFA faults are close to those for random faults. However, it is worth keeping in mind that the IFA faults are more significant since they represent realistic faults. Therefore, the similarity in the fault coverages does not mean that IFA should be disregarded. Rather, the undetected IFA faults should be focused on more than the undetected random faults.

For circuit B02, the fault coverage for IFA faults is consistently higher than for random faults. Also, the average fault coverage for IFA faults is always greater than or equal to that for random faults. One reason for

	Circuit			
	B02	*B06*	*B20*	*B22_1*
Number of nodes in circuit	49	99	12546	16381
Number of multi-node bridging faults extracted	27	22	96	164
Number of dominant-AND faults	62	53	234	420
Number of dominant-OR faults	62	53	234	420
Number of dominant-bridge faults	62	53	234	420
Number of wired-AND faults	27	22	96	164
Number of wired-OR faults	27	22	96	164
Extraction time:				
Parsing	<1min	<1min	6mins	14mins
Algorithm			<1min	<1min
Conversion to ADAMA			<1min	<1min

Table 4.1: Results of bridge extraction

this might be that the random faults generated by ADAMA include the inner nodes of complex gates whereas our implementation only considers the input and output signals of the complex gates as whole blocks. These inner nodes might be more difficult to diagnose and, therefore, result in a lower fault coverage. Another possible reason which is rather optimistic, is that ADAMA is simply better at detecting the more realistic faults.

In the end, the difference in fault coverage between IFA and random faults depends largely on the layout of the circuit simply because extraction of the IFA faults themselves depends on the layout.

	Circuit											
	B02			*B06*			*B20*			*B22_1*		
	Total injected	*Undetected*	*Fault coverage*	*Total injected*	*Undetected*	*Fault coverage*	*Total injected*	*Undetected*	*Fault coverage*	*Total injected*	*Undetected*	*Fault coverage*
Dom-AND:												
IFA	62	3	95	53	9	83	234	26	89	420	51	88
Random	62	18	71	53	6	89	234	36	85	420	49	88
Dom-OR:												
IFA	62	5	92	53	4	92	234	27	88	420	38	91
Random	62	23	63	53	6	89	234	46	80	420	45	89
Dom-bridge:												
IFA	62	1	98	53	0	100	234	8	97	420	12	97
Random	62	8	87	53	1	98	234	21	91	420	17	96
Wired-AND:												
IFA	27	0	100	22	0	100	96	4	96	164	3	98
Random	27	6	78	22	0	100	96	0	100	164	1	99
Wired-OR:												
IFA	27	0	100	22	1	95	96	5	95	164	4	98
Random	27	4	85	22	1	95	96	4	96	164	2	99
Total:												
IFA	240	9	96	203	14	93	894	70	92	1588	108	93
Random	240	59	75	203	14	93	894	107	88	1588	114	93

Table 4.2: Results of fault simulation and diagnosis

Chapter 5

Conclusion

The problem which was to be addressed in this thesis was obtaining a ranked list of realistic multi-node bridging faults to inject into the circuit. The problem was solved by implementing the IFA algorithm presented in [4]. The algorithm was explained and details regarding how it was implemented were also shown. The implementation allows the extraction of bridges involving an unlimited number of nodes. It also orders the faults in the fault list by their probability of occurrence, which allows the user to either take all the faults produced or select the most probable ones. Finally, the implementation was used to analyze some sample circuits and was compared with the random fault generation feature in ADAMA, after performing fault simulation and diagnosis. The obtained results were shown and explained in Section 4. It was shown that the heuristics used in ADAMA are more capable of detecting realistic faults than randomly generated ones. In addition to implementing the IFA algorithm, a new algorithm for polygon splitting was presented. The algorithm uses the corners of the polygons to perform the splitting and therefore is easily applicable to Boundary elements in GDS files. Combining this algorithm with the implementation allows circuit layouts with rectilinear polygons to be processed.

In addition to being used for diagnosis algorithm evaluation, the implementation can also be used to improve the manufacturability of circuits. This is done by first finding the problematic nodes in the circuit. After that, the layout of the circuit can be modified to make the areas related to these nodes more fault tolerant. This can reduce the effect of random spot defects on the circuit and can also be used to fix systematic proximity-related defects such as the electron beam proximity effect. Further accuracy can be achieved us-

ing this approach if the coordinates of the critical areas are extracted from the implementation. This will allow easier pinpointing of the problematic parts of the wires since nodes can span long lengths of wires and so do not provide an accurate physical location on the chip. The implementation can also be used to detect any occurrences of forbidden pitches.

Choosing the fault model

The IFA algorithm in the implementation provides a list of multi-node bridging faults. This allows the user to choose which fault model to apply, as mentioned previously, and in this implementation, a somewhat exhaustive test for several fault models was used. Further work regarding this area would be to find a way to choose the most realistic fault model to use for each fault.

Bibliography

[1] S. Holst and H.-J. Wunderlich, "Adaptive debug and diagnosis without fault dictionaries," in *12th IEEE European Test Symposium*, May 2007, pp. 7–12.

[2] ——, "A diagnosis algorithm for extreme space compaction," in *Proc. Design, Automation and Test in Europe Conference*, Apr. 2009, pp. 1355–1360.

[3] A. Jee and F. Ferguson, "Carafe: An Inductive Fault Analysis Tool for CMOS VLSI Circuits," in *11th Annual IEEE VLSI Test Symposium*, Apr. 1993, pp. 92–98.

[4] S. Zachariah and S. Chakravarty, "A Novel Algorithm for Multi-Node Bridge Analysis of Large VLSI Circuits," in *Proc. 14th International Conference on VLSI Design*, Jan. 2001, pp. 333–338.

[5] J. Shen, W. Maly, and F. Ferguson, "Inductive Fault Analysis of MOS Integrated Circuits," *IEEE Design Test of Computers*, vol. 2, no. 6, pp. 13–26, Dec. 1985.

[6] W. Maly, "Realistic Fault Modeling for VLSI Testing," in *Proc. 24th Conference on Design Automation*, Jun. 1987, pp. 173–180.

[7] A. Sreedhar and S. Kundu, "On design of test structures for lithographic process corner identification," in *Proc. Design, Automation and Test in Europe Conference*, Mar. 2011, pp. 1–6.

[8] A. Wong, "Microlithography: Trends, challenges, solutions, and their impact on design," *IEEE Micro*, vol. 23, no. 2, pp. 12–21, Apr. 2003.

[9] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*, ser. Electrical Engineering, Communications, and Signal Processing. IEEE Press, 1994.

[10] S. Zachariah and S. Chakravarty, "Extraction of Two-Node Bridges From Large Industrial Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 3, pp. 433–439, Mar. 2004.

[11] M. Berg, O. Cheong, and M. Kreveld, *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[12] E. McCreight, *Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles*, ser. CSL-80-9. Xerox, Palo Alto Research Center, 1980.

[13] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, ser. Texts and Monographs in Computer Science. Springer-Verlag, 1985.

[14] Z. Stanojevic and D. Walker, "FedEx - A Fast Bridging Fault Extractor," in *Proc. International Test Conference*, Nov. 2001, pp. 696–703.

[15] F. Goncalves, I. Teixeira, and J. Teixeira, "Integrated Approach for Circuit and Fault Extraction of VLSI Circuits," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov. 1996, pp. 96–104.

[16] The Perl Programming Language. [Online]. Available: http://www.perl.org/

[17] Cadence Encounter Digital Implementation System. [Online]. Available: http://www.cadence.com/products/di/edi_system/pages/default.aspx

[18] NanGate 45nm Open Cell Library. [Online]. Available: http://www.nangate.com/?page_id=22

[19] I.-L. Tseng. (2007) OwlVision tool. [Online]. Available: http://www.owlvision.org/