Implementation of a Design Rule Checker for Silicon Wafer Fabrication

by

Evren R. Ünver

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1994

Author _____
Department of Electrical Engineering and Computer Science
May 16, 1994

Certified by _____
Donald E. Troxel
Professor of Electrical Engineering
Co-Thesis Supervisor

Certified by _____
Michael B. Mc.Ilrath
Research Scientist
Co-Thesis Supervisor

Accepted by _____
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Implementation of a Design Rule Checker for Silicon Wafer Fabrication

by

Evren R. Ünver

Submitted to the
Department of Electrical Engineering and Computer Science

May 16, 1994

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

A table based design rule checker for silicon wafer processing has been implemented to verify the integrity of process flow recipes. Its goal is to enhance the safety of the operators and machines in the fabrication facilities while reducing the amount of time spent on the development and approval of process flows. The Design Rule Checker can be used either as an off-line process design aid, or as an on-line process monitoring tool. The first stages of integrating the Design Rule Checker into the graphical tree editor of the MIT Computer Aided Fabrication Environment (CAFE) have been completed. It is hoped that this new design rule checker will be both a powerful design tool for fabrication engineers as well as a useful utility for the software development personnel.

Co-Thesis Supervisor: Donald E. Troxel
Title: Professor of Electrical Engineering

Co-Thesis Supervisor: Michael B. McIlrath
Title: Research Scientist

Implementation of a Design Rule Checker for Silicon Wafer Fabrication

by

Evren R. Ünver

Submitted to the
Department of Electrical Engineering and Computer Science

May 16, 1994

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

A table based design rule checker for silicon wafer processing has been implemented to verify the integrity of process flow recipes. Its goal is to enhance the safety of the operators and machines in the fabrication facilities while reducing the amount of time spent on the development and approval of process flows. The Design Rule Checker can be used either as an off-line process design aid, or as an on-line process monitoring tool. The first stages of integrating the Design Rule Checker into the graphical tree editor of the MIT Computer Aided Fabrication Environment (CAFE) have been completed. It is hoped that this new design rule checker will be both a powerful design tool for fabrication engineers as well as a useful utility for the software development personnel.

Co-Thesis Supervisor: Donald E. Troxel
Title: Professor of Electrical Engineering

Co-Thesis Supervisor: Michael B. McIlrath
Title: Research Scientist

Contents

List of Figures

List of Tables

Chapter 1

1 INTRODUCTION

Since the creation of the first integrated circuit in 1960, there has been an ever increasing density of devices manufacturable on semiconductor substrates. The number of devices manufactured on a chip exceeded the generally accepted definition of *very large scale integration*, or VLSI (i.e. more than 100,000 devices per chip), somewhere in the mid-70's. This number is currently on the order of millions of devices per chip. Progress in VLSI manufacturing technology seems likely to continue to proceed in this manner, and even further reductions in the unit cost per function are projected. A simplified model of the processing required to fabricate integrated circuits using current technology can be seen in Figure 1-1.

With ever increasing automation and computer integration in the field of integrated circuit fabrication, the number and complexity of process steps required to manufacture a product have increased as well. As a result, it has become difficult to assess the overall effect of process steps and their inter-relations easily and quickly.

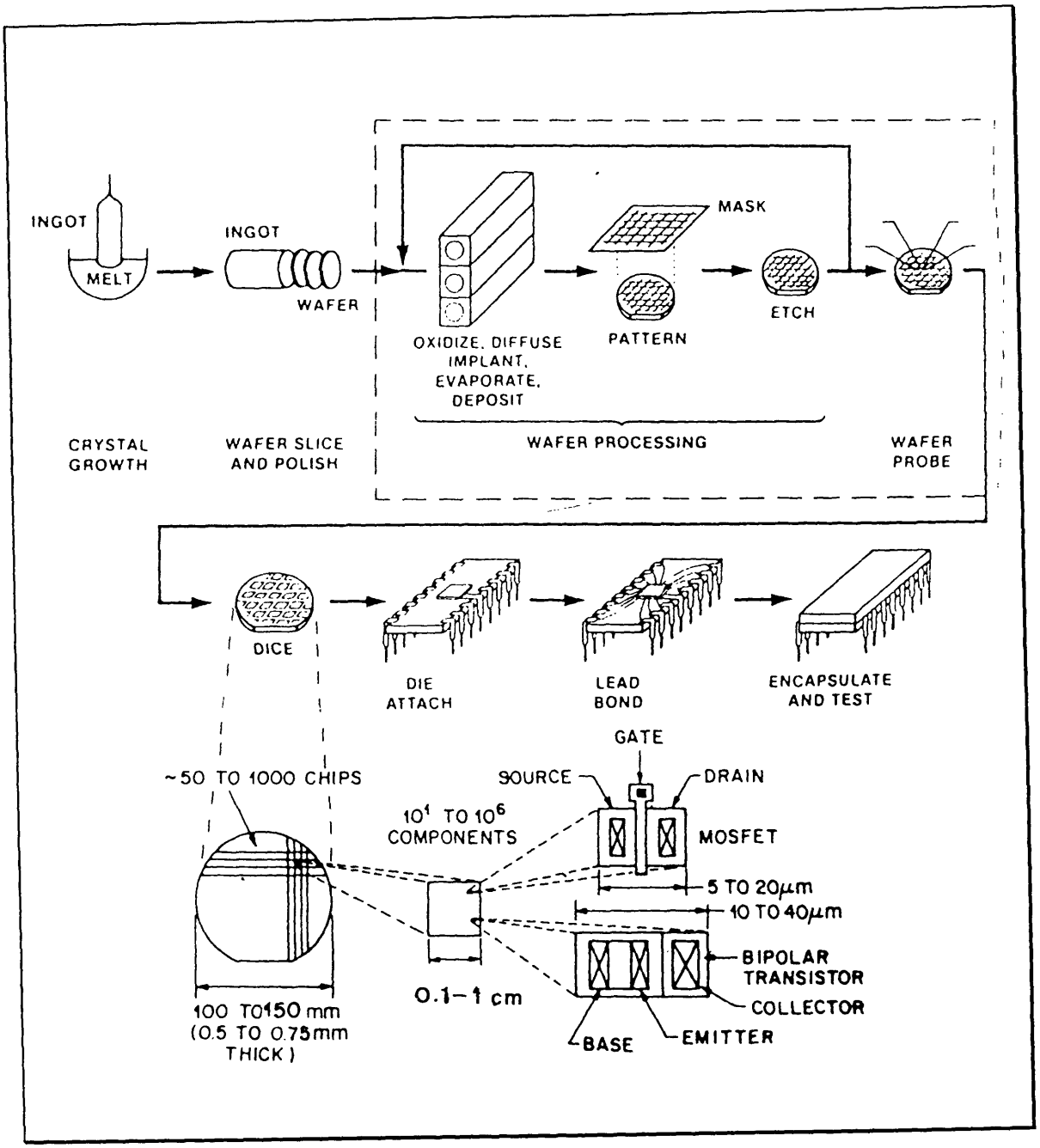


Figure 1-1: The fabrication process sequence of integrated circuits. Reprinted from Silicon Processing, by Wolf & Tauber [1].

As process steps have grown in complexity, they have also been broken down into smaller, modular building blocks, allowing them to be strung together to create larger processes. This layer of abstraction has also made it easy to overlook aspects of the underlying components. It is possible to inadvertently specify process flows that can cause damage to the wafers, or fabrication equipment. At the MIT Integrated Circuit fabrication facilities, there are currently no automated safeguarding mechanisms. To avoid potentially dangerous situations, as well as aid in the development of well-formed process flows, a Design Rule Checker is essential.

The goal of this thesis is to develop an extensible and general framework for design rule checking, and, using this framework, to implement the design rule constraints that have been identified through discussions with Linus Cordes, the director of the MIT semiconductor fabrication facilities.

1.1 Motivation and Rationale

In its simplest form, a Design Rule Checker can be used off-line, as an optional verification of the correctness of a process flow, before using it in processing. Eventually, it can evolve into an integral part of the design and manufacturing process, providing both interactive debugging help in the creation of process flows, and a measure of safety during the manufacturing, by monitoring dynamic fabrication conditions. This is especially useful in the cases where the processing recipe must be altered for various reasons during the processing. At this point, the Design Rule Checker can be used to verify the integrity and

validity of the new process. This will allow checks to be performed on each step of the process flow as it is executed (on-line monitoring), by evaluating each step in the context of the processing sequence undergone, blocking hazardous steps as necessary. By performing dynamic checking, more flexibility is gained from a processing point of view, as in situ changes can be made to the process flow without the risk of damaging wafers or equipment. This is especially important in light of the fact that it is possible to create processes "on the fly" in the fab by stringing together modular process building blocks.

By integrating such a design rule checker into the CAFE system, both users and the facilities will benefit from factors such as laboratory integrity and time saved in manual design rule checking. This will translate to improvements in the overall goal of fabricating wafers.

1.2 Background

The Computer Aided Fabrication Environment (CAFE) is a software system being developed at MIT for use in the fabrication of integrated circuits and microstructures. It is intended to be used in all phases of process design, development, planning and manufacturing of integrated circuit wafers [2]. CAFE is currently being used at the integrated circuit processing facilities of the MIT Integrated Circuit Laboratory, Lincoln Laboratories and Case Western University.

The CAFE manufacturing system is unique in the development of a Process Flow Representation (PFR) and its integration into actual fabrication operations. The motivation behind developing the PFR was to create a single, unified wafer processing representation in order to facilitate the integration of design and manufacturing in integrated circuit fabrication [3]. The PFR is a knowledge representation language, and is intended to represent information in a way that is not specific to any particular application. The PFR represents a process as a sequence of hierarchical operations to be performed on a group of wafers. This hierarchical representation can be visualized as a tree, with operations having child and parent operations associated with them. The syntax of the PFR is very similar to LISP.

The fabrication of wafers with a process represented as a PFR involves several steps [4]. A suitable PFR for the specified process must be created and installed in the CAFE system database. Wafer lots are then created, and "started", to create a task data structure that is isomorphic to the flow data structure. Actual machine operations are accomplished by instructions given to the operator and machines. Then data collected from the operation can be input into the database.

1.3 Design Rules

These are the process rules that have been identified through discussions with Linus Cordes, the director of the MIT semiconductor fabrication facilities, and implemented as part of the work. A brief explanation is given for each rule.

1) **No Photoresist in diffusion tubes**

Photoresist is an organic material. If wafers coated with Photoresist are exposed to a high temperature furnace operation the wafers will be ruined and the furnace tube will be contaminated.

2) **No Photoresist in RCA clean**

Photoresist will contaminate the RCA clean bath.

3) **No Photoresist in Nitride wet etch**

Photoresist will contaminate the Nitride wet etch bath.

4) **No Photoresist in Varian metallization system**

The tubes that are used for metal deposition will be contaminated by Photoresist.

5) **No RCA clean after metal deposition**

The metal will be destroyed and the RCA clean bath will be contaminated.

6) **No Piranha clean after metal deposition**

The metal will be destroyed and the Piranha (H_2SO_4) clean bath will be contaminated.

7) **No Oxide etch after metal deposition**

The metal will be destroyed and the Oxide etch bath will be contaminated.

8) **No Nitride wet etch after metal deposition**

The metal will be destroyed and the Nitride wet etch bath will be contaminated.

9) **Metallized wafers in tube B7 or B8 only**

This is a facility specific rule. To reduce metal contamination metallized wafers are only processed in furnace tubes B7 and B8.

10) **No fused quartz wafers in the Varian metallization system**

The Varian metallization system is equipped with optical sensors, which require an opaque wafer surface for proper operation. As fused quartz wafers are not opaque they cannot be used in the Varian.

11) **Must have Oxide etch prior to Nitride wet etch**

The nitride layer cannot be etched properly if there is a native oxide on the wafer (a thin layer grows spontaneously even at ambient conditions). Therefore a Nitride wet etch must always be proceeded by an Oxide etch.

12) **Wafers must go through Coater prior to Stepper**

The order of the lithography steps must be maintained, and after wafers are coated with Photoresist they must go directly into the Stepper (where the photoresist is exposed to UV light). The only step allowed in between is an inspection.

13) **Wafers must go through Developer following Stepper**

After the Stepper, the next step must always be the Developer. As in rule (12), the only step allowed in between is an inspection.

14) **BPSG deposition must be followed by BPSG flow**

Borophosphosilicate glass (BSPG) films tend to be hygroscopic and unstable, and should therefore be flowed immediately following deposition. The only step allowed in between is an inspection.

15) **Wafers must have oxide etch following Phosphorous deposition prior to drive**

Before entering a furnace tube the residual native oxide from the Phosphorous deposition must be removed.

16) **Wafers must have pre-metal clean prior to metallization if preceded by a photo lithography step**

Some test structures, such as certain capacitors, are built without any lithography steps. In this case, there is no need to perform a pre-metal clean before metallization. However, at all other times metallization must be preceded by a pre-metal clean.

17) **Tube A2 must be used only for thin oxide growth (<500Å) on high resistivity (>1Ω.cm) silicon**

This is another facility specific rule. Tube A2 is used solely for gate oxide growth, and should not be used for other purposes.

18) **No wafers with high phosphorus concentration in tubes A1 or A2**

This is another facility specific rule. These tubes are used for growing high quality oxides, and Phosphorus contamination is to be avoided.

Chapter 2

2 PREVIOUS WORK

The ideas of computer aided wafer fabrication and using a unified process flow representation are relatively new, and consequently there has not been much research in this specific area regarding design rule checking. However, the concepts encountered are similar to some other research areas, and it is important to review them in order to benefit from work already done.

2.1 VLSI Design Rule Checking

There has been a lot of interest in design rule checking for the circuit layout level of integrated circuit manufacturing. The need for and benefits of automation in this field were recognized early on, and consequently there have been many different approaches to this problem. Early implementations relied on various software based systems, as in Baird [5], followed by hardware assisted design rule checkers, such as Seiler [6], Blank et al. [7] and Longhead and McCubbrey [8], providing gains due to the custom built hardware. More

recently, however, due to the general increase in computing power, and the more flexible approach, most design rule checkers have migrated back to software based solutions [9].

In general, these systems typically rely on various geometric pattern matching algorithms to perform the rule checking.

2.2 Wenstrand's Model for Specification, Simulation and Design of Semiconductor Fabrication Processes

In his 1991 Ph.D. thesis [10], Wenstrand develops an object-oriented model for the specification, simulation and design of semiconductor fabrication processes. In doing so, he investigates the idea of using explicit design constraints to construct objective functions for optimization of process parameters. By associating design goals with the manufacturing process specification, in addition to providing a statement of intended effect, the verification of the correct behavior of a module is simplified.

According to Wenstrand, the process simulation and verification needs to be approached through qualitative and quantitative simulation. Using quantitative simulation, actual numerical design goal parameters can be compared to values obtained from a numerical process simulator, such as SUPREM III. By comparing the results to minimum and maximum acceptable values for the processing step, the process flow can be monitored.

Through qualitative simulation, the objective is to check process *sequences* to flag potential safety, manufacturing, and equipment hazards, without requiring actual numerical processing data. This is the general approach that will be followed in this thesis.

2.3 Constraint Based Programming

In Steele's thesis on constraint based programming languages [11], a *constraint* is defined to be a "declarative statement of relationship" or "a computational device for enforcing the relationship". Similar ideas will be used in developing a design rule checker that is driven by the rules that are implemented by the system. For example, one constraint could be that "A furnace step must be preceded by a cleaning operation or a furnace step". In this case, a process sequence that violated this constraint would cause an error.

The difference from Steele's view of constraints is that in the design rule checker the constraints will not be used to actually *repair* a dangerous process flow, but will only point to the problem.

2.4 Supervisory Control Theory

In Supervisory Control Theory, the notion of the controllability of a problem is investigated. According to Balemi et al. [12], a plant model can be seen in two ways, as in Figure 2-1.

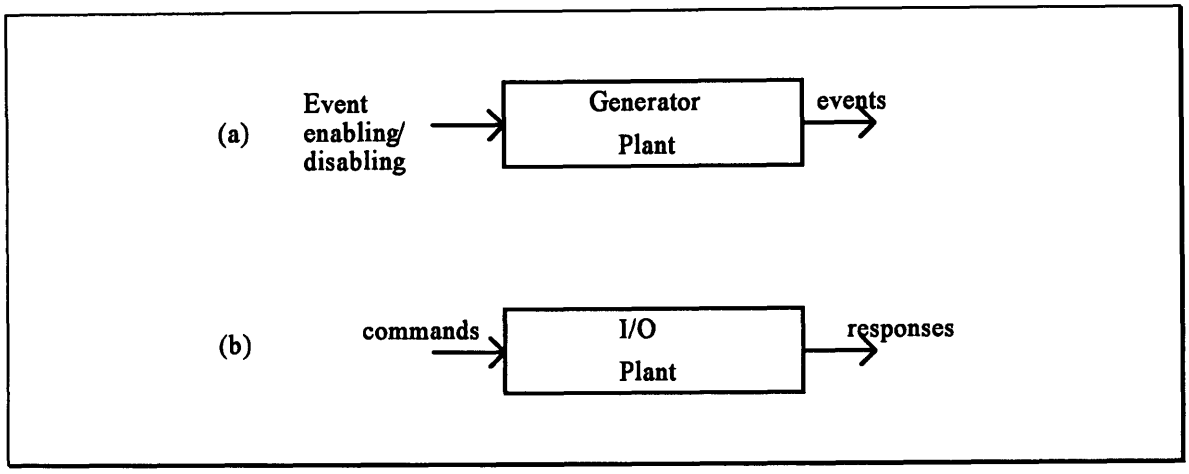


Figure 2-1: (a) Generator plant and (b) Input/Output plant

In the first case (Figure 2-1(a)), a plant event "generator" produces events "wildly", and the only way to affect the behavior of the plant is by enabling and disabling the controllable events. In this scenario, the plant alone *schedules* the occurrence of both controllable and uncontrollable events.

However, to better model the plant, an input-output perspective is required, as in most real systems events do not occur spontaneously, but only as *responses to commands* (Figure 2-1(b)). Therefore, by connecting the plant with a controller to complete a feedback loop, events can be directed in the desired way (Figure 2-2). In this case, the inputs to the system, or the commands, would be the process flow (recipe) for the operation.

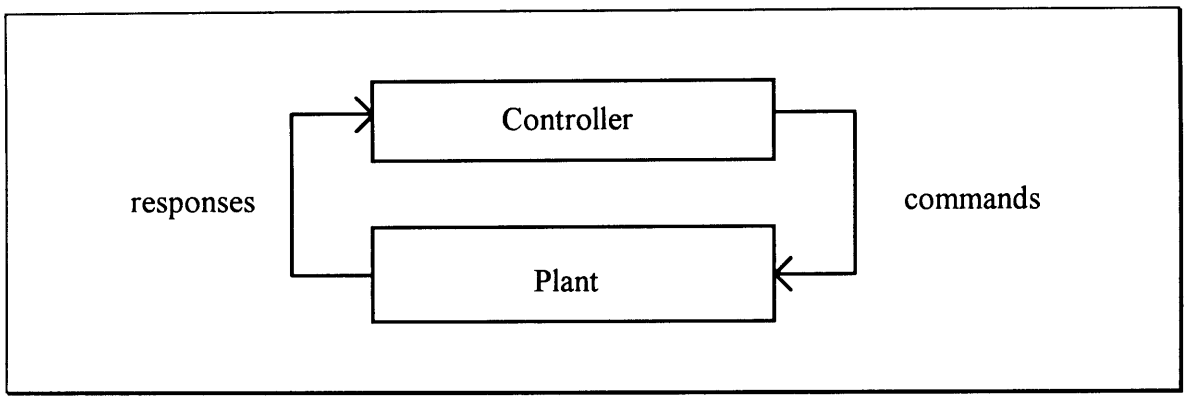


Figure 2-2: Plant with Controller feedback loop

To make the system safe, a Supervisor is connected between the Controller and the Plant (Figure 2-3). The Supervisor serves a dual purpose. First, by combining it with the Controller, a supervised (checked) process is obtained. Second, by combining it with the Plant, a supervised Plant is obtained, ensuring on-line safety. Similarly, the Design Rule Checker can be thought of as a Supervisor.

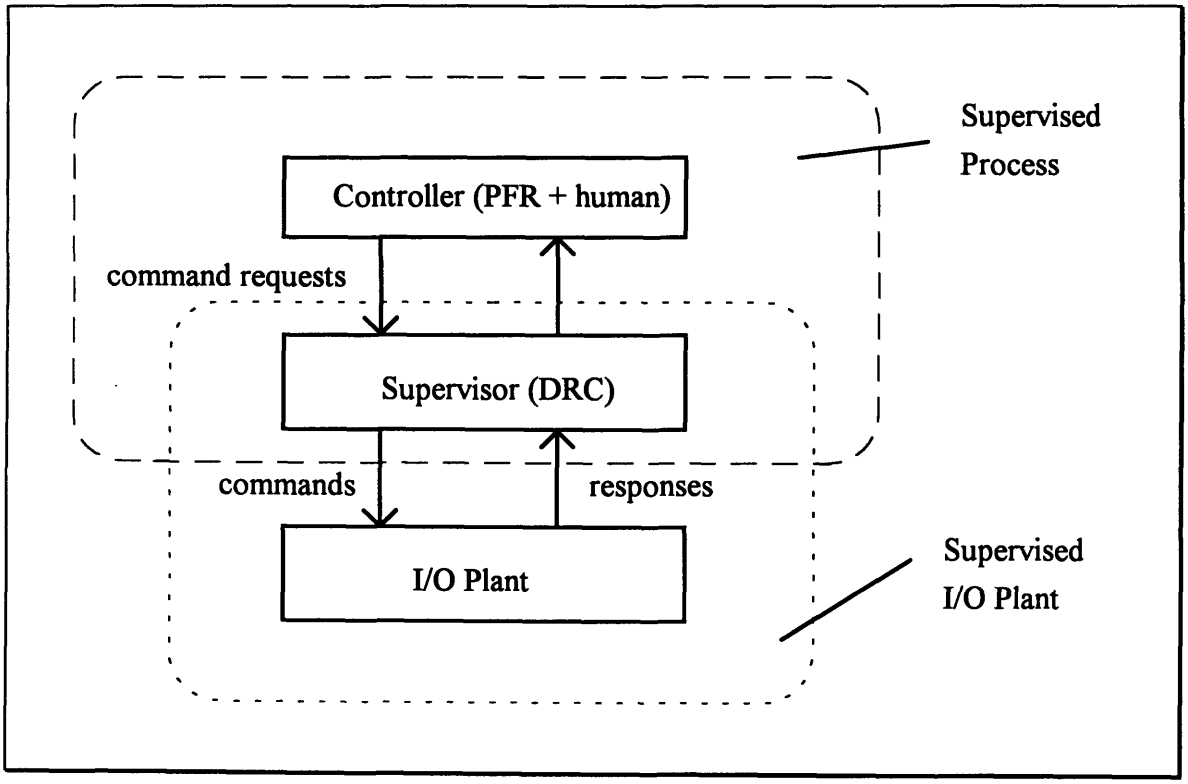


Figure 2-3: Supervisory control scheme

2.5 Duane Boning's Design Rule Checker

The outline for a collection of independent design rule checks for CAFE was proposed by Duane Boning in 1989. These included fabrication safety rules as well as internal consistency checks. However, only two checks were implemented, and they were never integrated into the CAFE system. They were based on the textual representation of the PFR, and were intended for single wafer operations.

2.6 The Hitachi Process Flow Validation System

A Rule-Based VLSI Process Flow Validation system with macroscopic process simulation has been developed at the Hitachi Central Research Laboratory [13]. The starting point of this project is that it is difficult to catch process flow mistakes on run sheets as they can become long and complicated. An example where a cleaning step is missed between ashing and oxidation steps is shown in Figure 2-4.

The rules considered are grouped into four categories. These are:

1) **Process window:**

   These are the simplest type of rules. They do not require any prior knowledge of wafer state or process sequence. Only the possible or allowable conditions within a single process step are needed. Examples of this type of rule include furnace temperature and etching gas species.

2) **Process sequence:**

   These rules require knowledge about the process sequence and conditions. Examples of this type of rule include pre-cleaning, annealing and post-cleaning.

3) **Wafer-process constraint:**

   Wafer-process constraints are defined as the constraints between wafer state and process or equipment. Here the wafer state includes information on macroscopic wafer structure, that is, the kind of substances and contamination existing on a Si wafer, and

their geometries (thickness, patterned/non-patterned) and properties (implanted/non-implanted, baked/non-baked).

To check for these rules, wafer state and process condition information is necessary. These rules include "Contaminated wafers should not be loaded into clean furnaces" and "Resist removal condition depends on resist properties (thickness, hardened, etc.)".

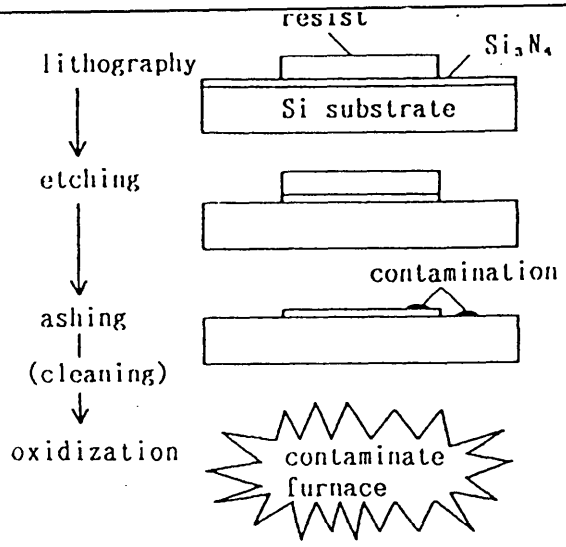
4) **Optimum conditions**

This last group of process rules concerns the optimum sequence and conditions to fabricate the intended VLSI structure and characteristics. Some of this knowledge depends on the total process type, such as CMOS, bipolar or BiCMOS. To perform this type of a rule check, a variety of information is needed, such as process flow, intended structure and characteristics, purpose of experiment, and results of other lots, in addition to detailed process and device simulations.

This type of rule checking is only discussed conceptually, and is not implemented.

Rules belonging to the first three groups have been implemented in a dialect of Common LISP on a HITAC M680 mainframe computer. The system has about 180 design rules.

The system is used as an off-line process design aid. The main beneficiaries are process experts who used to check designed flows and can now perform higher level checks.



(a)

wafer / Step	1	2	3	4	5	6	7	8	9	10
lithography	○	○	○	○	○	○	○	○	○	○
etching 1	○	○	○	○					○	
2					○	○	○	○		○
inspection									○	○
ashing	○	○	○	○	○	○	○	○	○	○
ion 1	○				○					
doping 2		○				○				
3			○				○			
4				○				○		
cleaning	○	○	○	○	○	○	○	○		
Oxidization	○	○	○	○	○	○	○	○	○	○

(b)

Fig. 1. Example of process flow mistake. Contaminated wafers will be loaded into clean furnace. (a) Contaminated wafers loaded into clean furnace. (b) Same mistake as (a) in a real process flow. Each wafer is processed in circled steps. Wafer 9 and 10 should be cleaned before oxidation.

Figure 2-4: Example of process flow mistake. Reproduced from article by Funakoshi and Mizuno published in the IEEE Transaction on Semiconductor Manufacturing [13].

Chapter 3

3 CIDM SOFTWARE OVERVIEW

The Computer Integrated Design and Manufacturing (CIDM) group at MIT is focused on developing and demonstrating critical elements of a framework for the design and manufacture of advanced integrated circuits. One of the key software applications that have resulted from this effort is CAFE, the Computer Aided Fabrication Environment. CAFE provides a framework for many different types of fabrication applications, including process design, development, simulation, laboratory scheduling, wafer lot management, and manufacturing of integrated circuit wafers. CAFE currently provides day to day support to research and production facilities at MIT with both flexible and standard product capabilities.

The CIDM software environment was developed to provide a single, unified system for wafer fabrication. All CIDM software programs share the same database interface layer and information representation. This environment is also intended to be extensible, easily modifiable, and modular, having well defined interfaces between separate components.

As seen in Figure 3-1, the CIDM System Architecture can be broken down into three levels. The lowest level is the CIDM Infrastructure Architecture, comprising an object oriented database model which is implemented in a layered manner on top of a relational database. The database schema is based on GESTALT [14], an object oriented, extensible data model. GESTALT is a layer of abstraction which provides a mapping of user defined objects onto existing database systems, in this case INGRES™ [1], a relational database.

The second layer is the CIDM Data and Tool Integration Architecture level. This level includes the conceptual schema and models used to represent the integrated circuit manufacturing domain if CAFE, and the user and programmatic interfaces to the various higher level applications.

The third layer is the Applications level, which is made up of the separate software programs for scheduling, fabrication support, data collection, design rule checking, etc.

[1] INGRES™ is a trademark of Ingres Corporation.

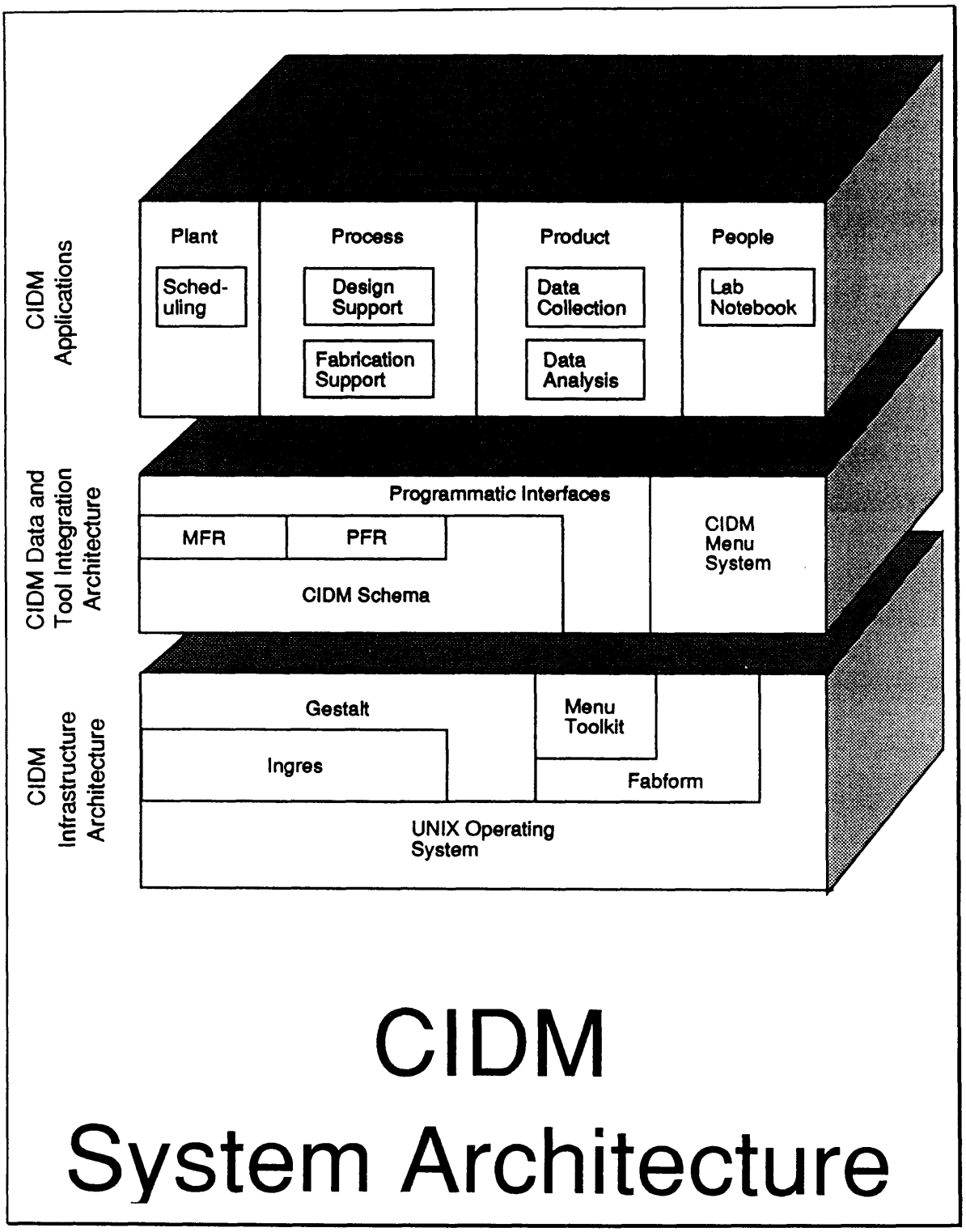


Figure 3-1: CAFE system architecture

3.1 GESTALT

GESTALT is an expressive database programming system developed at MIT [14]. It provides the top level database interface to the CIDM software. Its purpose is to provide a simple database interface to an application program in the program's host language. This way, the application programs are shielded from the details of the underlying database's query language.

In GESTALT, objects are modeled as belonging to a certain type and having certain attributes. They may also contain and/or share other objects. GESTALT currently provides selectors, mutators, constructors and inverse fetch functions in Lisp and C. The LISP interface uses the Common LISP Object System (CLOS) [15]. These accessors are implemented as CLOS methods.

The selector functions allow the user to get a handle on a certain database object using attributes such as parent/child relationships. The mutator functions allow the user to change the attributes associated with an object. The constructor functions allow the user to create a new object of the given type. The inverse fetch functions allow the user to get a handle on database objects using attribute values, such as the *name*.

This provides for very simple and straightforward integration into the application code. Some example GESTALT methods in LISP are:

SELECTOR:	(task-subtasks task)
MUTATOR:	(setf (task-subtasks task) tasks)
CONSTRUCTOR:	(make-task)
INVERSE FETCH:	(tasks-with-name task)

Table 3-1: Gestalt Methods

In the current CAFE system the underlying database, where the data is actually stored, is INGRES. However, GESTALT is designed to be flexible enough to be used with other databases as well.

3.2 Schema

There are over 70 different GESTALT data types that have been defined in the schema. There are basically two types of data: Domain Specific, and non-Domain Specific. Non-Domain Specific data do not have instance identity; i.e. they are "values", not "objects". These include integers, floating point numbers, strings, booleans, etc. Domain Specific data, on the other hand, are GESTALT objects with attributes. These attributes in turn may be other GESTALT types. Wafers, Wafersets, Tasks, Machines are examples of Domain Specific data. Each Domain Specific instance of an object has a unique identifier.

3.3 Process Flow Representation

The CAFE manufacturing system is unique in the development of a Process Flow Representation (PFR) and its integration into actual fabrication operations. The motivation behind developing the PFR was to create a single, unified wafer processing representation in order to truly facilitate the integration of computers into wafer processing.

The Process Flow Representation was developed at MIT by Michael McIlrath and Duane Boning to meet the requirements of such a unified wafer processing representation [3]. The PFR is a knowledge representation language, and is intended to represent information in a way that is not application specific, but general.

The PFR represents a process as a sequence of hierarchical operations to be performed on a group of wafers. This hierarchical representation can be visualized as a tree, with operations having child and parent operations associated with them.

3.3.1 PFR Syntax and Structure

The syntax of the PFR is very similar to LISP. Some of the important features are:

Define The `define` construct allows a symbol to be bound to a value or procedure. The form is:

(define <name> <form>)

The <form> value can either be a simple constant or a more complicated form. For example:

```
(define OxideGrowthTime 3600)
(define GateOxTube "tubeA1")
(define gate-oxide
  (operation
    (:change-wafer-state
       (:deposit :material :oxide
                 :thickness (:microns 1)))))
```

The define construct can also accept arguments, thus creating parameterized definitions (also called "functions" in this context). The form is:

```
(define (<name> [{<parameter-name> |
                  (<parameter-name parameter-default>)}])
  {<forms>})
```

An example of a parameterized definition is

```
 (define (HMDS-prime (recipe 1))
  (operation
     (:machine "HMDS")
     (:settings :material "HMDS" :recipe recipe)
     (:time-required (:minutes 45))))
```

HMDS-prime could be called as illustrated below:

```
(define calling-operation
  (operation
    (:body
      (HMDS-prime :recipe 3)   ; called with recipe = 3
      ...                      ;        < OR >
      (HMDS-prime)             ; called using default value
```

Operation The `operation` construct actually defines a wafer processing operation and its attributes. It is currently interchangeable with the flow construct. The basic form for an operation is:

```
(operation
  [(:doc <documentation-string>)]
  [(:version <version-entries>)]
  [(:permissible-delay <delay>)]
  [(:advice <advice>)]
  [(:time-required <time-required>)]
  [(:body <body>)]
  [(:change-wafer-state <change-wafer-state>)]
  [(:treatment <treatment>)]
  [(:machine <machine>)]
  [(:instructions <instructions>)]
  [(:readings <readings>)]
  [(:settings <settings>)]
  [(:opset <opset-name>)]
)
```

Note that `:doc`, `:version`, `:permissible-delay`, etc. are the attributes for the operation. The attribute `:body` defines sub-operations for this operation. The following example defines an operation named "`example1`" with values for the `:doc`, `:version`, `:time-required`, and `:body` attributes. In this example, the values for the `:body` attribute are in terms of their defined operations, but another unnamed operation could be defined instead.

```
(define example1
  (operation
     (:doc "Example of operation construct")
     (:version 1.1)
     (:time-required (:minutes 30))
     (:body
       subexample1
       subexample2
       subexample3)))
```

If One of the more powerful features in PFR is that of conditionals. The if construct is a special form and allows branching on condition:

```
(if <condition>
 <then-clause>
 [<else-clause>])
```

An example of using the if construct is:

```
(define (furnace-rampdown-treatment start-temperature
                                    (anneal-time (:minutes 30)))
  (sequence
    (if (>? anneal-time 0)
      (:thermal :temperature start-temperature
                :time anneal-time :ambient :N2))  ; Anneal
    (:thermal :temperature start-temperature :ambient :N2
              :time (:minutes (/ (- start-temperature 800) 2.5))
              :temp-rate -2.5)  ; Ramp-Down
    (:thermal :temperature 800
              :time (:minutes 20) :ambient :N2)))  ;Stabilization
```

There are many other important constructs in the Process Flow Representation. More information on the PFR may be found in [16].

3.3.2 PFR Fabrication

The fabrication of wafers with a process represented as a PFR involves several steps. A suitable PFR for the specified process must be created and installed into the database. This is accomplished by "evaluating" the textual representation of the PFR, and creating persistent flow objects, which are then stored in the database.

The tree structure of these objects reflects the hierarchical decomposition of the PFR encoding. Wafer lots are then created, and "started", to create a task data structure. Actual machine operations are accomplished by instructions given to the operator and machines. Then data collected from the operation can be input into the database.

3.3.3 Process Flow Tree vs. Task Flow Tree

The actual fabrication operations are not driven by the flow tree structure. The process flow tree contains all of the information pertaining to a specific procedure, but does not contain information such as the status of the operation in progress. When the processing of a certain lot of wafers begins, a *task flow tree* specific to that flow and wafer lot is created. Therefore in most respects the process flow tree and task flow tree are isomorphic data structures (Figure 3-2).

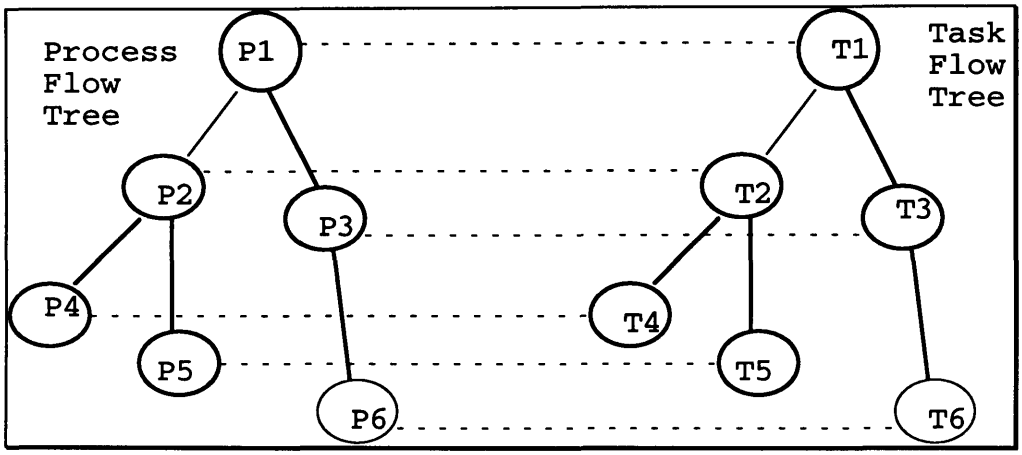


Figure 3-2: Process Flow Tree and Task Flow Tree

Differences primarily arise when the task flow tree associated with a lot that is being processed is edited "on the fly", during processing (for example to change the temperature or duration of a furnace operation). In this case, the edited node of the task flow tree no longer points to the old node in the isomorphic process flow tree, but may point to a different, unconnected process flow tree instead.

Chapter 4

4 APPROACH TO DESIGN RULE CHECKING

4.1 Finite State Machine Representation

The way to visualize the behavior of the process rules is to think of each design rule as a finite state machine. This approach was proposed by Michael McIlrath in an unpublished memo [17].

In this description based on discrete automata, a conceptual "supervising automaton" (or monitor) controls the state transitions. A state transition takes place in the machine under supervision only if allowed by the supervisor.

For example, the state diagram for rule (1), "No Photoresist in diffusion tubes" is shown in Figure 4-1.

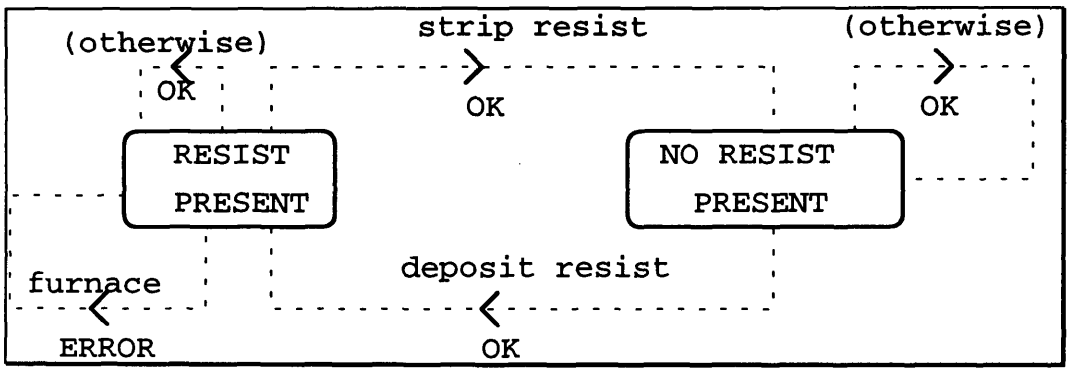


Figure 4-1: FSM approach to Design Rules

4.2 Algorithm for design rule checking

The basic algorithm used for design rule checking is quite straightforward. The approach is to construct a system that checks for design rule constraints at each subsequent processing step, based on the current *state* of the wafer. By *wafer state* here we refer not only to the simulated or measured process effects at the present node, but also to the process history.

Rules may include "sanity" checks, that is, obvious rules which wouldn't necessarily cause any damage (such as putting wafers through the Stepper without Coating them first) as well as "safety rules", which if ignored would damage equipment and wafers (i.e. photoresist in diffusion tubes). Each design rule is a separate entity, and can be developed

and applied independently. This way, it is not necessary to sequentially run multiple design rule checking programs to check for multiple rules.

The basic steps are as follows:

4.2.1 Determine process wafers

At each node of the process flow tree, the wafers to be processed are identified. The actual mechanisms for doing so depends on whether a process flow tree or task flow tree is being checked, and will be discussed in further detail in Chapter 5.

The function of identifying the process wafers is specifically isolated from the rest of the system, as there is not a complete agreement to the issue of what the lowest common denominator for wafer processing is. This confusion arises from the fact that wafers are hierarchically grouped into three levels, each represented by a GESTALT data type. The simplest is the type "wafer". It denotes a single wafer, and its properties. Among these properties is the *wafertype*, which contains information about the starting material. The second type is a "waferset". A waferset is a grouping of wafers which have undergone the same processing and is intended as a means of simplifying the representation and providing a useful layer of abstraction. The third type is a "waferlot". A waferlot is the set of all of the wafers that are specified in the processing. See Figure 4-2 for examples of these three data types.

Since the PFR is very flexible by nature, there are no enforced rules regarding the uses of wafersets, and their specification. Therefore the wafers specified in the PFR may in fact represent more than one individual wafer, if they are going through the same process steps.

In other words, from the point of design rule checking there is no difference between a flow with a single wafer, or a dozen wafers, as long as they all go through the same processing. The key issue is that the waferset abstraction must be used in a consistent manner.

```
#<WAFER 35720040> is an instance of class #<Gestalt-Class
WAFER 34104620>:
 The following slots have :INSTANCE allocation:
 %ENTITY          12221200
 ID               "splits,5"
 TYPE             #<WAFERTYPE PP+NTYPE 56517600>
 LASERID          "F1"
 TIME_BROKEN      NIL
```

(a)

```
#<WAFERSET splits-F1 34750560> is an instance of class
#<Gestalt-Class WAFERSET 34105200>:
 The following slots have :INSTANCE allocation:
 %ENTITY     12205944
 NAME        "splits-F1"
 TIME        #<TIMEINTERVAL 35720060>
 WAFER       (#<WAFER 35720040>)
 PARENT      NIL
 ADVICE      NIL
```

(b)

```
#<LOT splits 34750700> is an instance of class #<Gestalt-
Class LOT 34074240>:
 The following slots have :INSTANCE allocation:
 %ENTITY                    12080352
 ID                         "splits"
 LABUSER                    #<LABUSER evren 34410000>
 CREATIONDATE               "03/24/94"
 WAFERSET                   (#<WAFERSET splits-F1 34750560>
                             #<WAFERSET splits-E5 34750600>
                             #<WAFERSET splits-F6 34663640>)
 STATUS                     "ACTIVE"
 TASKS                      (#<TASK GEN_1-EVREN 34750720>)
 STATUS_COMMENT             NIL
 RESPONSIBLE_USER           #<LABUSER evren 34410000>
 TASKSTRUCTUREMODIFIED      #<TIME 35720240>
 TASKSTATUSMODIFIED         NIL
 REPORT                     #<REPORTCACHE splits Traveller
35720220>
 PRIORITY                   "NORMAL"
 READYTASKS                 (#<TASK RCA-CLEAN 35720200>)
 PROCESSINSTANCE            #<PROCESSINSTANCE 35720160>
```

(c)

Figure 4-2: (a) Wafer, (b) Waferset, (c) Waferlot object instances

4.2.2 Check processing and update state

To maintain an accurate description of the wafer state at each processing node, it is necessary to track the processing steps that are relevant to the design rules we are checking for. For example, consider rule (4), "No Photoresist in Varian metallization system". For this rule to function properly, the steps that deposited Photoresist and etched Photoresist (if they existed) would have to have caused a change in the *state* of the wafers that were identified as being processed at that step.

This is accomplished through what Wenstrand calls *Qualitative Process Simulation* in his thesis [10]. Similar to the rule checking, the process simulation is table driven as well, so it is possible to add or subtract the process steps that are monitored by the system, depending on the rules being checked.

After the effective processing has been simulated, and the relevant aspects of the change in wafer states have been recorded, this new wafer state is stored at that node of the processing tree (Figure 4-3).

In effect, this provides a qualitative summary of the processing that those wafers have undergone. This is very convenient, as once a flow has been checked in its entirety, each node contains a description of the wafer state up to that point. Now, if a modification is made in one of the sub processes, it is not necessary to check the flow from the beginning.

```
#<TASK METAL-DEPOSITION 35714460> is an instance of class
#<Gestalt-Class TASK 34102100>:
 The following slots have :INSTANCE allocation:
 %ENTITY                    12277168
 NAME                       "METAL-DEPOSITION"
 LOT                        #<LOT eutest1 35720740>
 WAFERSETS                  (#<WAFERSET eutest1-w1 35720720>
                             #<WAFERSET eutest1-w2 35720700>
                             #<WAFERSET eutest1-w3 35720660>
                             #<WAFERSET eutest1-w4 35720640>)
 STATUS                     "PLANNED"
 MACHINES                   NIL
 SCHEDULED_TIMEINTERVAL     NIL
 FLOW                       #<PROCESSFLOW METAL-DEPOSITION
35714040>
 SUBTASKS                   (#<TASK ORGANIC-CLEAN&ETCH
35720500>
                             #<TASK INSPECT-THICKNESS
35720460>
                             #<TASK SPUTTER-DEPOSIT
35720360>)
 PLANOPINST                 NIL
 NEXT_LEAF_TASK             NIL
 ADVICE                     "(:OPSET \"mvarian\" :WAFERSTATE
        ((\"w1\" . #S(WAFER RESIST T METAL NIL))
         (\"w2\" . #S(WAFER RESIST T METAL NIL))
         (\"w3\" . #S(WAFER RESIST T METAL NIL))
         (\"w4\" . #S(WAFER RESIST T METAL T))))"
```

Figure 4-3: Wafer state at each process node

4.2.3 Apply design rules

As mentioned earlier, the rule checking at each process node is table driven. Each rule in the table is a constraint associating wafer state information and process step information. The constraint is active just when the process step information matches the attributes of the process step specified in the PFR. Thus only steps that are significant to the particular design rule are taken into account.

If a constraint is not satisfied, an error is signaled. Unsatisfied constraints thus *block* further execution of the process. If a design rule is violated, a warning message is displayed, stating the design constraint, the predicate that caused it to fail, the wafer affected, the name of the flow and the back-trace of the processing sequence to point out where the problem occurred. See Figure 4-4 for an example of the warning messages.

4.2.4 Repeat in the body

The same steps described previously are repeated for every node of the processing tree. The tree structure is in effect linearized, as the processing is inherently sequential. The order in which the tree is traversed is top-down and left-to-right (Figure 4-5).

```
CAFE>(rules2 b)

Warning: NO-METAL-IN-RCA and METAL-PRESENT on wafer "w1" in
flow: "RCA-CLEAN"

TRACE: (EUTEST2 unnamed FLOW WELL-DRIVE RCA-CLEAN)

Warning: NO-METAL-IN-TUBES and METAL-PRESENT on wafer "w1"
in flow: "FURNACE-OPERATION"

TRACE: (EUTEST2 unnamed FLOW WELL-DRIVE FURNACE-OPERATION)

Warning: NO-METAL-IN-PIRANHA and METAL-PRESENT on wafer "w2"
in flow: "PIRANHA-CLEAN"

TRACE: (EUTEST2 unnamed FLOW PIRANHA-CLEAN-OPERATION
PIRANHA-CLEAN)

Warning: NO-METAL-IN-OXIDE and METAL-PRESENT on wafer "w3"
in flow: "OXIDE-BOE-ETCH"

TRACE: (EUTEST2 unnamed FLOW NITRIDE-WET-ETCH GENERIC-WET-
ETCH OXIDE-BOE-ETCH)

Warning: NO-METAL-IN-OXIDE and METAL-PRESENT on wafer "w4"
in flow: "OXIDE-BOE-ETCH"

TRACE: (EUTEST2 unnamed FLOW NITRIDE-WET-ETCH GENERIC-WET-
ETCH OXIDE-BOE-ETCH)

Warning: NO-METAL-IN-NITRIDE and METAL-PRESENT on wafer "w3"
in flow: "NITRIDE-WET-ETCH-OPERATN"

TRACE: (EUTEST2 unnamed FLOW NITRIDE-WET-ETCH GENERIC-WET-
ETCH NITRIDE-WET-ETCH-OPERATN)

Warning: NO-METAL-IN-NITRIDE and METAL-PRESENT on wafer "w4"
in flow: "NITRIDE-WET-ETCH-OPERATN"

TRACE: (EUTEST2 unnamed FLOW NITRIDE-WET-ETCH GENERIC-WET-
ETCH NITRIDE-WET-ETCH-OPERATN)

"EUTEST2"

CAFE>
```

Figure 4-4: Sample warning messages

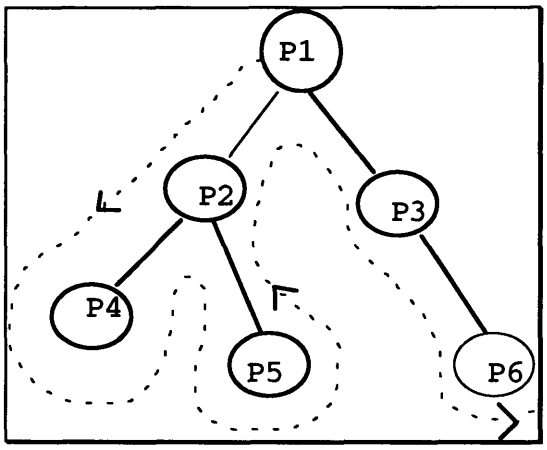


Figure 4-5: Threading the tree

This is somewhat different from simply traversing the fringe tasks of the tree structure (which is what is done to determine the next fabrication operation in CAFE, as in Figure 4-6).

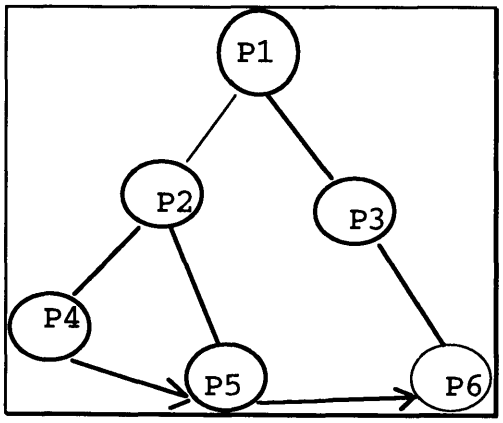


Figure 4-6: Following the fringe tasks

It is possible that the parent node contains change in wafer state or treatment information that is not specified in the child nodes. Therefore by choosing the first method, all of the processing is accounted for.

4.3 Important Features

In this section various aspects of the design rule checking system are discussed.

4.3.1 Use of Database Objects

The PFR is a platform that may yet change in the future. So to be independent of the current PFR syntax, and to be able to implement on-line monitoring, it was decided to use the database representation of the flow objects instead of the textual representation (Figure 4-7), providing greater flexibility.

```
(fl-load "/usr/cafe/pfr/lib/lib-loc.fl")
(fl-library :database)
 (fl-load "utils.fl")
 (define EUTEST3
  (flow
   (:body
    nitride-wet-etch
    (oxide-boe-etch :ACID "7-1boe" :TIME 15 :TANK 2 :ADD-
TIME 900)
    (resist-develop :RECIPE 20)
    (resist-expose :MASK CD :MASK-ID "CD"
             :DSWJOB "NEW DA CWR1" :INSTRUCTIONS"")
    (resist-coat :INSTRUCTION "")))))
```

Figure -4-7: Textual Process Flow

The database representation of an installed flow tree is shown Figure 4-8. The attributes such as the sub-flows or the change in wafer state of the process flow are easily accessible via the GESTALT layer commands described earlier in section 3.1.

```
#<PROCESSFLOW EUTEST3 34751260> is an instance of class
#<Gestalt-Class PROCESSFLOW 34077540>:
 The following slots have :INSTANCE allocation:
 %ENTITY              12053296
 NAME                 "EUTEST3"
 VERSION              NIL
 TIMEREQUIRED         NIL
 SUBFLOWS             (#<PROCESSFLOW NITRIDE-WET-ETCH
34661340>
                       #<PROCESSFLOW OXIDE-BOE-ETCH 34751020>
                       #<PROCESSFLOW RESIST-DEVELOP 34751000>
                       #<PROCESSFLOW RESIST-EXPOSE 34750760>
                       #<PROCESSFLOW RESIST-COAT 34750740>)
 INSTRUCTIONS         NIL
 TREATMENT            "NIL"
 DOC                  NIL
 CHANGEWAFERSTATE     "NIL"
 SETTINGS             "NIL"
 READINGS             "NIL"
 MACHINE              "NIL"
 ADVICE               "(:FILE
\"/amd/garcon/a/evren/pfr/eutest3.fl\" :NAME \"EUTEST3\")"
 WAFERSETNAMES        "NIL"
```

Figure 4-8: Installed Process Flow

4.3.2 Splits and Joins

In actual processing, especially in a research environment, there are often many splits in a wafer lot. Frequently the operator will use certain settings for one batch of wafers, take measurements, and based on the empirical results modify the recipe to obtain better results for subsequent wafers.

Especially in long and complicated flows, the addition of complex splits can make it very difficult for the operator to be fully aware of all the steps that all of the wafers are going through, and the inter-relations of the process steps (see Figure 4-9 for an example of a flow with many splits). This is exactly the kind of situation where the Design Rule Checker will be of great assistance to the operators, as it can maintain and update the state of each wafer in the lot.

```
(define GENSIMOX
  (flow
   (:wafers ("948" "960" "964" "967" "968" "969" "B3" "B6"
"C0"))
   (:doc "Ge implanted NMOS on SIMOX with Tsi splits")
   (:version
    (:modified :number 1.0 :by "Quan Xiong" :date "Nov. 15,
1993"
            :what "Create PFR"))
   (:body
    (flow
     (:wafers ("960" "964" "967" "968" "969"))
     (flow
      (:wafers ("960" "964" "967"))
      (dfield1-1k :instructions "Wet O2 Time=20mins, recipe
114, use Tube B2 if possible" :names "1100A-Field-Oxide"))
...
...
...
```

Figure 4-9: Excerpt from a flow with many splits

4.3.3 Use of the Task Flow Tree structure

The Design Rule Checker can be used either with the process flow structure, or the task flow structure. The task flow tree is initially isomorphic to the process flow tree, but contains information more specifically related to the wafer lot being processed. See Figure 4-10 for an example of the task flow structure.

This reflects the two distinct modes that the Design Rule Checker is meant to be used in: the first, utilizing the process flow structure, can be thought of as a process design aid, where it is used completely off-line, in the initial development stages of a process flow. By checking the flow, the general *well-formedness* of the flow can be verified, and the user can be warned if an illegal or incorrect process flow has been specified so that any corrections that are needed can be made.

In the second mode, by utilizing the task flow structure, it is possible to use the design rule checker in a more active way, during the actual processing of the wafers. This way a higher level of laboratory integrity can be maintained, and depending on the policies of the fabrication facility, design rule checking can be made mandatory before each step.

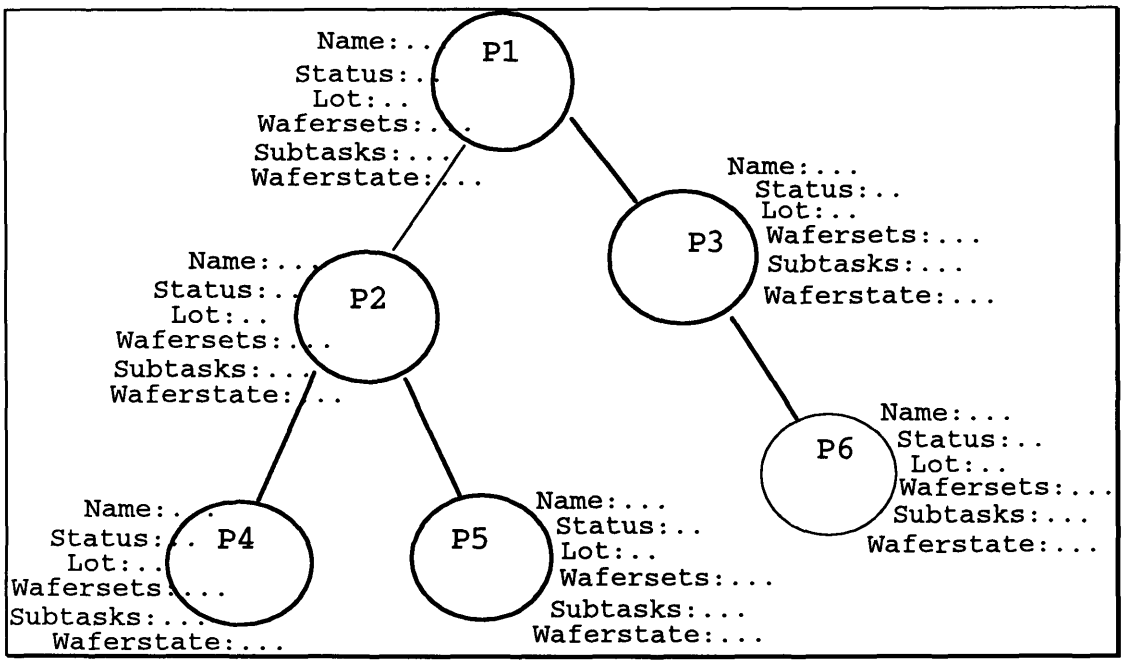


Figure 4-10: Example Task Flow structure

Chapter 5

5 OVERVIEW OF THE SOFTWARE

This chapter presents the software organization of the Design Rule Checker. One of the main design principles of the Design Rule Checker was to separate the code as much as possible into logical modules. For example, wafer-state manipulation functions are independent of the design rule functions, and so on. Also rules themselves are separated from the code. The advantage of this approach is that future improvements in one module can go on without having to modify the other modules. Also, it makes the structure of the program easier to understand for future developers.

5.1 Wafer representation

5.1.1 DRC-Wafer

A specialized representation of the wafer is stored as a CLOS [15] object to keep track of the necessary state information. It is similar to a standard wafer type, but it has slots for Photoresist and Metal states. These are binary states, which are either true or false. An example drc-wafer (design-rule-check wafer) is seen in Figure 5-1.

```
#<DRC-WAFER 56516000> is an instance of class #<Standard-
Class DRC-WAFER 35712420>:

 The following slots have :INSTANCE allocation:

 TIME              2977487318

 CONCENTRATION     0.0

 DOPANT            NIL

 ORIENTATION       :|100|

 MATERIAL          :SI

 METALIZED         NIL

 RESIST            NIL
```

Figure 5-1: Example drc-wafer

There are accessors defined that allow easy manipulation of the drc-wafer and its attributes:

`make-drc-wafer`	Create a new drc-wafer with default values
`drc-wafer-resist`	Access the resist slot
`drc-wafer-metal`	Access the metal slot

Table 5-1: Drc-wafer accessors

It is possible to change the value of the slot by using constructs such as:

```
(setf (drc-wafer-resist <drc-wafer>) t)
```

to change the value of the slot.

5.1.2 Printed Representation

The drc-wafer structure is convenient as it provides a layer of abstraction and ease of access to the storage and representation of the wafer state. However, it has one drawback: It is not of a readable form in LISP, and consequently cannot be coerced into a string.

This proves to be important, because as described in Chapter 4, it is necessary to store the wafer state information at each node of the process tree. The simplest place to store this wafer state information is in the "advice" slot of the task structure. The advice slot provides for general extensibility and is used by different programs to store various items of information, such as the location of a traveller file (a summary of the processing steps), or the name of a Statistical Process Control file. By convention, the advice slot is a property list, that is, a list of name-value pairs. Each program reads and writes the properties it knows about and leaves the rest untouched.

Since the advice field is a string, it is not possible to save the drc-wafer form there directly. The solution is to use a structure which has been designed to be a simplified form of the state information stored in the drc-wafer, and automatically has a readable printed representation. The printed representation of a wafer structure is:

```
(#S WAFER RESIST NIL METAL NIL)
```

```
(xform-drc <drc-wafer>)
```

This function simply takes a CLOS wafer, which contains certain wafer state properties (Figure 5-1) and converts to the structure, maintaining the same state information. Thus a readable representation of the same information is obtained.

Note that the #S printed representation is used only to store the wafer state at the process nodes. The internal calculations are based on the drc-wafer CLOS object, which is stored in memory during the execution of the Design Rule Checker program.

The only time the #S printed representation is converted back to the drc-wafer object is during the initialization of the wafer states at the beginning of the design rule checking. If the flow has been previously checked (and consequently has wafer states stored at the nodes in #S structure representation) then the drc-wafers are created using the saved (#S representation) wafer state.

5.1.3 The Wafer List

The state of each wafer being processed in the lot is stored in the global variable *wafer-list*, which looks something like:

```
CAFE>*wafer-list*
(("w1"
  . #<DRC-WAFER 56517320>)
 ("w2"
  . #<DRC-WAFER 56517300>)
 ("w3"
  . #<DRC-WAFER 56517260>)
 ("w4"
  . #<DRC-WAFER 56517240>)
 ("w5"
  . #<DRC-WAFER 56517560>))
```

Figure 5-2: Wafer list

At each node, after the specified subset of wafers undergoes the "Process Simulation", the changes in the wafer state are stored back here.

5.2 Operation Wafers

The wafers that are to be operated on at each node of the process tree are stored in a stack, called *op-wafers* (a global variable). The wafers of the current node are always at the top of the stack, and by default the bottom of the stack is the list of all the wafers included, or the wafer lot.

The way the *op-wafers* stack is updated depends on whether the Design Rule Checker is being used to check process flows or task flows. For task flows, the process wafers for

each node are explicitly specified, therefore it is trivial to obtain them and push them on the stack.

For process flows, however, the only indication comes from the `(:wafers ...)` attribute in the flow. If this attribute does not exist, then it is assumed that the operation wafers are the same as of the parent process. Put another way, in a process flow without any splits in it, there would likely be no wafers specified at all in the PFR.

Therefore, while checking task flows, it is possible to obtain the initial lot of wafers directly from the task tree, in the case of process flows it is necessary to have the user specify them manually. When design rule checking process flows the user is first asked to input the list of wafers:

```
"Please input list of wafers to be processed [ex: (w1 w2)]"
```

This way the initial wafer lot is established, and operation wafers for subsequent nodes can be established by inheriting wafers from the parent process if none are explicitly specified.

In the case of checking process flows, the program also verifies that the wafer list input by the user are actually specified in the PFR.

5.3 Updating the Wafer State

The mechanism for performing the simple qualitative process simulation is as follows:

the processing steps that are relevant to the rules we are interested in checking are identified. For the set of rules used here, only depositing photoresist, etching photoresist and depositing metal are of importance (etching metal is not relevant because in the current group of rules that have been implemented all of the metal-related rules only care if a wafer has ever been metallized or not).

The wafer state operations are represented in the program in table form as function pairs made up of a process predicate and a change wafer state operation, as in Table 5-2.

Process predicate	**Change wafer state operation**
`deposits-resist`	`deposit-resist`
`etches-resist`	`etch-resist`
`deposits-metal`	`deposit-metal`

Table 5-2: Wafer State

After the operation wafers for a processing node are determined, these wafer steps are applied sequentially, and if the process predicate returns true for the processing node, then the change wafer operation is invoked, which modifies the wafer state for the current wafers.

Due to the simple table driven structure, it is easy to modify the wafer state operations that are tracked.

5.4 Program representation of the rules

The heart of the Design Rule Checker program, the design rules, are quite similar in format to the wafer state operations used to update the wafer state. The program representation of the list of rules that were identified in Chapter 1 are seen in table form in Table 5-3.

Process step predicate	**BLOCKING wafer state predicate**
1) resist-not-allowed	resist-present
2) no-resist-in-RCA	resist-present
3) no-resist-in-Nitride-Wet-Etch	resist-present
4) no-resist-in-Varian	resist-present
5) no-metal-in-RCA	metal-present
6) no-metal-in-piranha	metal-present
7) no-metal-in-oxide	metal-present
8) no-metal-in-nitride	metal-present
9) no-metal-in-tubes	metal-present
10) no-quartz-in-varian	failed
11) need-oxide-before-nitride	failed
12) must-have-coater-before-stepper	failed
13) must-have-developer-after-stepper	failed
14) must-have-bpsg-flow-after-dep	failed
15) must-have-oxide-after-phos	failed
16) must-have-pre-metal-clean	failed
17) only-high-resistivity-in-a2	failed
18) only-thin-ox-in-a2	failed
19) cannot-enter-tube-A1-or-A2-after-Phos.	failed

Table 5-3: Design Rules

The process step - blocking wafer state design rule pairs that are implemented in this table can be divided into two major categories. The first group of rules (rules 1-8) depend on the *wafer state* that has been simulated and recorded for the wafers. The second group of rules (rules 9-19) depend only on the *sequence* of the processing steps. These rules are independent of the processing the wafers have undergone, or the *wafer state* they are in.

Some of the rules, like rule (9), "Metallized wafers in tube B7 or B8 only" are specific to the MIT Integrated Circuit Laboratory (ICL), and wouldn't be directly applicable to a different fabrication facility. However, most of the rules have been implemented in a more general fashion, based on processing fundamentals, and are facility-independent.

Note that it may appear that rules like (10) and (17) appear to be dependent on wafer state. However, the relevant properties (whether it is made of quartz, or high resistivity silicon in these cases) of the wafers do not change with processing. Therefore they do not need to be tracked, and are determined from the wafertype attribute which specifies the starting material (see section 4.2.1).

It should be also noted that the cost of implementing and integrating new design rules is low. As can be seen in the program listings in Appendix B, each design rule module is quite simple in itself. One of the major goals of this project was to develop a general and extensible framework for design rule checking, allowing for portability. Thus it is relatively easy to customize the program for specific needs.

Mechanisms for locating the previous and next leaf tasks already exist in the system. These can be commonly used by all of the design-rule modules. Consequently it is very simple to design multi-level sequence related rules using simple logical operators such as AND, OR, etc. By factoring in wafer state information even more advanced rules can be devised with ease.

Chapter 6

6 INTEGRATION INTO CAFE

To make the Design Rule Checker a truly useful tool, it must be presented in a form that is intuitive as well as simple to use. If it can only be used by people familiar with the intricacies of LISP its appeal will be severely limited.

To this end, the first steps have been taken to integrate the Design Rule Checker into the Computer Aided Fabrication Environment.

6.1 Command-line control

The most rudimentary interface to the Design Rule Checker is through the LISP Listener in CAFE. For process flows, the Checker requires only the name of the installed flow, and for task flows, the Checker requires the unique id. of the task. So a process flow or a task can be checked by simply entering:

```
(rules <flow name>)    or,

(rules2 <task eid>)
```

The wafer state operations and the safety rules that are used by the program will be the values of the variables *wafer-state* and *safety-rules*. The output will be any resulting warning messages.

This method of using the program is very basic, and suitable for debugging purposes. However it requires some knowledge of the underlying structure.

6.2 The Tree Editor

The Tree Editor is a program that was developed at MIT by Albert Woo as a generic graphical tree editor for wafer processing using the Tcl/Tk command language and X11 toolkit [18]. It is meant to ease the process of composing and editing wafer fabrication sequences using the PFR by providing a graphical interface.

It is currently possible to skip, delete, add or modify tasks using the Tree Editor. Adding a Design Rule Checker option to the Tree Editor was a natural extension, as the already existing functions for selecting and manipulating the task tree objects fit well with the style in which the Design Rule Checker was intended to be used.

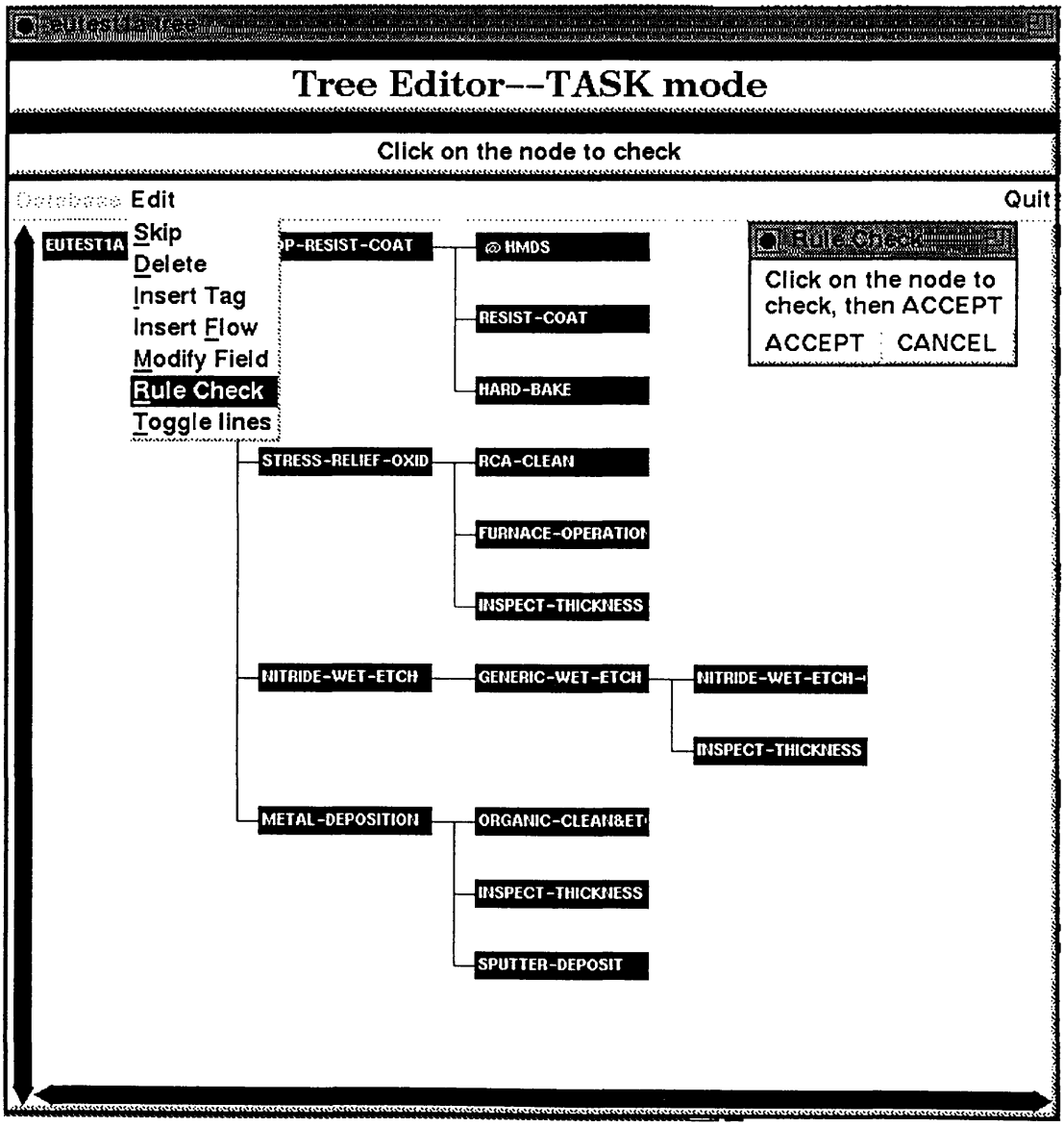


Figure 6-1: Example of task tree

The communication link between the Tree Editor and the Design Rule Checker is very simple. The only information passed from the Task Tree is the unique identifier of the task to be checked.

To invoke the Design Rule Checker program through the Tree Editor, the user must first load the task tree to be checked. The Rule Check option is located under the Edit menu, and clicking on it will prompt the user to pick the node to start the checking from. The first time the check is performed, it must be done from the root of the tree, in order to ensure correct wafer state information. Now, if the user decides to modify certain parameters, insert or delete steps, it is possible to only check the branches of the tree that will be affected by the changes without having to recompute everything.

6.3 Automatic checking

Another, more integrated way in which the Design Rule Checker can function is through automatic checking of the process flows that are being fabricated.

The obvious places this can be done are at "Start-Lot" time, when a process flow and a wafer lot are merged to create a task tree and begin fabrication, and at "Operate-Machine" time, i.e., before each machine operation is performed.

These issues depend mainly on the policies and goals of the fabrication facility.

Chapter 7

7 CONCLUSION

The Design Rule Checker project as described in this thesis is available to the CIDM personnel for testing. The initial reactions from the faculty, staff and graduate students who have seen the program have been very positive and encouraging. It is considered that the Design Rule Checker will be a useful tool. However, the real test will be whether it actually proves to be useful in the day to day operations of the fab.

7.1 Contributions

A rule based design rule checker has been developed for use with the MIT Process Flow Representation. Previous to this program the only way to check the validity of a process flow recipe was to do it manually, in a time consuming and tedious fashion. From the point of the Computer Aided Fabrication Environment project, the Design Rule Checker is a natural extension and fulfills an existing requirement.

One of the most important aspects of this program which is different from other design rule checkers is that it is possible to use it for on-line fabrication monitoring. The Process

Design Aid described in Wenstrand's thesis and the Hitachi Rule-Based Process Flow Validation System are limited in their scope in that they can only be used in the initial creation stages of the process flow development operation.

By enabling on-line fabrication monitoring, more realistic design rule checking is possible, as every modification made up to the point of actually fabricating the device is accounted for. In addition, the possibility of compromising the actual safety and integrity of the facilities and personnel are reduced to a minimum.

At the MIT Integrated Circuits Laboratory, only roughly half of the lots that are processed are processed using the Process Flow Representation. At Lincoln Labs, however, all lots use the PFR. Over the past year, graphical tools such as the Task Tree Editor and the Flow Tree Editor have made the user interface to developing flows much simpler. This has encouraged people previously intimidated by the LISP-like appearance of process flows to use the PFR. It is hoped that the process design aid aspect of the Design Rule Checker will contribute to the usefulness and ease of PFR-based fabrication.

7.2 Future work

In its current state the Design Rule Checker is an independent program that, when called with a process flow tree or a task flow tree, outputs a series of warning messages. A working model and framework for design rule checking within the CAFE system have

been demonstrated. However, this does not mean that this project should be viewed as completed. There are many improvements and extensions that should be considered.

7.2.1 Optimization

There is room for improvement and optimizations in the program code. Though acceptable for most applications if considered a one-time computational cost, depending on the complexity of the flow, the number of splits & joins, and the number of rules being checked, the computation can take a long time. For example, the Defect Array (DA) flow, which is considered to be the longest flow implemented under the PFR at the MIT ICL, takes around ten minutes to check on a Sun SPARC station 10 under average load.

It is typically difficult to determine which operations take the most time to compute without performing careful benchmark analyses. If one tries to optimize the code simply by attacking the areas that are assumed to be slow, chances are that the initial guesses were wrong. Having said this though, *possible* improvements might come from minimizing and caching the database accesses, and simplifying the algorithm for updating the wafer state for multiple wafers going through the same processing.

7.2.2 Further Integration into CAFE

As described in Chapter 6, it is possible to invoke the Design Rule Checker from within the Task Tree Editor. This is an example of the direction that should be followed in further integration into the CAFE system.

A simple graphical front end needs to be implemented to select the basic parameters for the design rule check. This would include a browser for the rules that were being checked, their descriptions, etc. For example, if a person only processes silicon wafers it is not necessary to check for rules such as (10), "No fused quartz wafers in the Varian metallization system". Hooks should be placed in the Flow Tree Editor so that the rule checker can be invoked from there as well.

To take full advantage of the on-line fabrication monitoring, performance-functionality trade-offs need to be investigated to determine the optimal method for design rule checking. Some issues include whether the rules are checked at "Operate machine" time, or "Start Lot" time, etc.

7.2.3 Expanded functionality

As it exists now the Design Rule Checker has a library of 18 rules. In comparison, the Hitachi Process Flow Validation System has over 180, including highly specialized rules that take into account factors such as differences in hardness, thickness and other properties of materials. Obviously there are many more rules that can be added to the Design Rule Checker's existing ones, that will further increase the program's functionality. As the rules were separated from the body of the program in a logically distinct way, it is quite easy to add more rules to the system.

Future work can also include more aggressive functionality in the system. Currently the design rule checking is only a passive mechanism which issues warning messages. In Steele's thesis on constraint based programming, constraints are also seen to have an active role in addition to a declarative role. In this sense, the Design Rule Checker could be expanded to actually change and repair flows that did not pass the rules.

Appendix A
DEVELOPER'S GUIDE

The Design Rule Checker has been developed in an extensible and open-ended manner. Consequently the difficulty of developing new rules to add to the existing library is relatively low. The specifics of the information required are presented in this section.

A.1 How to add a rule

It is very straightforward to add a new rule to the Design Rule Checker. It is only necessary to append the new rule to the existing ones in the *safety-rules* table. Detailed knowledge of the program is not necessary.

A.1.1 The Structure of the Safety Rules

The program representation of the rules are stored in the global variable *safety-rules*, as described in section 5-4. Each rule consists of a process step predicate and a wafer state operation predicate that blocks (in effect violates) the rule.

The process step predicate and the wafer step predicate are functions that have been defined to check for a certain rule. The process step predicate is called with the current flow-node that is being checked. For example, for rule (1), "No photoresist in diffusion tubes" the process step predicate is `resist-not-allowed`. To check this rule, the function

```
(resist-not-allowed <task/flow>)
```

is called at each node of the tree. This function checks whether the operation at that node involves a diffusion tube. If not, it returns `NIL`, and the next rule is called. If it is indeed a diffusion tube operation, then it returns `TRUE`.

When a process step predicate returns a value of `TRUE`, the corresponding wafer step predicate is automatically called. The wafer step predicate is called with the current wafers being processed. In this case:

```
(resist-present <wafer>)
```

will be invoked. This checks the current *state* of the operation wafer, and if it returns `TRUE`, then the rule has been violated, and an error message will be generated.

The error message is generated by combining the names of the process step predicate and wafer step predicate along with wafer and flow location information, therefore it is advisable to remain consistent in naming the functions to define the rules.

To incorporate the rule into the list of rules to be checked, it is sufficient to simply append the new rule to the list of current safety rules. The order is not important.

A.1.2 The Structure of the Wafer State operations

The wafer state operations used to track the qualitative process simulation undergone by the wafers are structured very similarly to the safety rules. They consist of a process predicate and a change wafer state operation. The process predicate is a function called with a flow node, and the change wafer operation is a function called with a wafer representation. To simulate metal deposition on a wafer, for example, the function

```
(deposits-metal <task/flow>)
```

would be called. Similarly, if it returns TRUE, then the function to modify and update the wafer state will be called:

```
(deposit-metal <wafer>)
```

As with the safety rules, to add new wafer state operations it is sufficient to append the new process predicate/change wafer operation pairs to the current list (*wafer-state*).

A.2 Additional Wafer State

In the future if there is a need for rules that require wafer state information that is not currently tracked (i.e. other than resist and metal), the `drc-wafer` CLOS object can be modified to meet the demands.

The current representation is derived from the `simwafer` class:

```
(defclass drc-wafer (simwafer)
  ((resist :initform nil :initarg :resist :accessor drc-wafer-resist)
   (metalized :initform nil :initarg :metal :accessor drc-wafer-metal)))
```

The corresponding printed representation can be modified in an analogous manner.

Appendix B

PROGRAM CODE

```
(in-package :cafe)
(require :fl "fl")

(export '(rules))

 ;;;
 ;;; (Adapted from Duane Boning's drc-resist)
 ;;;


(defun rules2 (flow-op)
  (declare (special *wafer*)        ;needed by core flow evaluator
           (special *operation-trace*)
           (special *the-flow*)
           (special *has-seen-photo-step*)
           (special *wafer-list*))
  (setf *wafer* (make-drc-wafer))
  (setf *op-wafers* (list (get-wafers2 flow-op)))
  (if (task-waferstate flow-op)
      (setf *wafer-list* (initialize-wafer-list flow-op))
    (setf *wafer-list* (make-wafers)))
  (setf *operation-trace* nil)
  (setf *the-flow* flow-op)
  (setf *has-seen-photo-step* nil)
  (rules-interp flow-op))

;; We use a specialized representation of the wafer to keep track of a
;; few pieces of state information needed for the design rule checks.

(defclass drc-wafer (simwafer)
  ((resist :initform nil :initarg :resist :accessor drc-wafer-resist)
   (metalized :initform nil :initarg :metal :accessor drc-wafer-metal)))

(defun make-drc-wafer (&rest args &key resist &allow-other-keys)
  (declare (ignore resist))
  (apply #'make-instance 'drc-wafer args))
```

```
;; Uses the following information from the flow representation:
;;   Resist state
;;    :cws :deposit     - to add resist
;;    :cws :etch  - to remove resist


(defun rules-interp (op-object)
  (declare (special *operation-trace*))
  (let ((op-name (task-name op-object)))
    (if op-name (push op-name *operation-trace*))
    (cond ((null op-object) nil)
          (t
;; get the list of wafers, or pass down previous
          (push (get-operation-wafers2 op-object) *op-wafers*)
; check if wafers actually specified initially
;         (check-if-wafers-specified)     ;don't need if used under task-
mode
          (check-wafer-steps op-object)
          (check-op-loop op-object (first *op-wafers*))
          (update-advice-waferstate op-object) ;add waferstate to advice
slot
;; go into the body, if it exists
          (if (task-subtasks op-object)
              (dolist (op-part (task-subtasks op-object))
                    (rules-interp op-part)
                    (pop *op-wafers*)))))
    (if op-name (pop *operation-trace*))))



;;
;;  LOOP FOR CHECKING *WAFER-STEPS*
;; ====================================

(defun check-wafer-steps (form)
  (let ((i 0))
    (loop
     (if (= i (list-length *wafer-steps*)) (return))
     (let ((op (nth i *wafer-steps*)))
       (check-wafer-process form op (first *op-wafers*)))
     (setf i (1+ i)))))

(defun check-wafer-process (form op wafers)
  (let ((process-result (funcall (step-predicate op) form)))
    (if process-result
      (update-wafer-processing op wafers))))

(defun update-wafer-processing (op wafers)
  (let ((i 0))
    (loop
     (if (= i (list-length *wafer-list*)) (return))
     (if (member (car (nth i *wafer-list*))
             wafers :test #'string-equal)
       (funcall (step-wafer-op op) (nth i *wafer-list*)))
     (setf i (1+ i)))))



;;
```

```
;;  LOOP FOR CHECKING *SAFETY-RULES*
;; ==================================

(defun check-op-loop (form wafers)
  (let ((i 0))
    (loop
     (if (= i (list-length *safety-rules*)) (return))
     (let ((rule (nth i *safety-rules*)))
       (check-operation form rule wafers))
     (setf i (1+ i)))))

(defun check-operation (form rule wafers)
  (let ((i 0))
    (loop
     (if (= i (list-length *wafer-list*)) (return))
     (let ((the-wafer (nth i *wafer-list*)))
       (if (member (car the-wafer)
                   wafers :test #'string-equal)
         (let ((process-mesg (funcall (rule-process-predicate rule)
                                      form)))
           (if process-mesg
             (let ((wafer-mesg (funcall (rule-wafer-predicate rule)
                                        the-wafer)))
               (if wafer-mesg
                   (progn
                   (format t "Warning: ~A and ~A on wafer ~S in flow:
~S~%"
                           (rule-process-predicate rule)
                           (rule-wafer-predicate rule)
                           (car the-wafer)
                           (task-name form))  ;was fl-name
                   (format t "TRACE: ~A~%~%"
                           (reverse *operation-trace*)))))))))
     (setf i (1+ i)))))



;Used only for the Flow-Mode (not Task-Mode)
(defun get-wafers ()
  (declare (special *op-wafers*))
  (format t "Please input list of wafers to be processed [ex: (w1
w2)]~%")
  (setf *op-wafers* (list (read))))

; for Task-Mode
(defun get-wafers2 (form)
  (declare (special *op-wafers*))
  (setf *op-wafers* (get-operation-wafers2 form)))


(defun make-wafers ()
  (declare (special *wafer-list*))
  (setf *wafer-list*
      (mapcar #'(lambda (x) (cons x (make-drc-wafer)))
            (first *op-wafers*))))

;use this to init. the *wafer-list* if processing starts somewhere down
the
;tree where the wafer properties may have changed
(defun initialize-wafer-list (task)
```

```
  (let* ((waferstates (task-advice-waferstate task)))
    (setf *wafer-list* (mapcar #'initialize2 waferstates))))

(defun initialize2 (tagged-s)
  (let* ((id (car tagged-s))
       (ss (cdr tagged-s)))
    (cons id
        (make-drc-wafer :resist (wafer-resist ss)
                    :metal (wafer-metal ss)))))


(defun get-operation-wafers2 (op-object)            ; gets wafers from
tasks
  (let* ((waferset (task-wafersets op-object))
       (wafers (mapcar #'waferset-wafers waferset))
       (wafers2 (flatten2 wafers))
       (laserids (mapcar #'wafer-laserid wafers2))
       (laserids2 (remove-duplicates laserids)))
    laserids2))


;Used only for Flow-Mode
(defun check-if-wafers-specified ()
  (let ((j 0))
    (loop
     (if (= j (list-length (first *op-wafers*))) (return))
     (let ((all-wafers (first (reverse *op-wafers*))))
       (if (not (member (nth j (first *op-wafers*))
                  all-wafers :test #'string-equal))
         (progn (format t "Warning: Wafer ~A not specified
initially.~%~%" (nth j (first *op-wafers*)))
              (format t "TRACE: ~A~%~%" (reverse *operation-trace*)))))
     (setf j (1+ j)))))



;;
;; #S wafer state for the task-advice slot representation
;;-------------------------------------------------------

(defstruct wafer resist metal)

(defun yap (x)
       (cons x (make-wafer)))

(defun make-them-wafers (task)
  (mapcar #'yap (get-operation-wafers2 task)))

(defun task-advice-waferstate (task)
  (let* ((advice (task-advice task)))
    (advice-slot-value advice :waferstate)))


;convert the drc-wafer representation to the #S notation w/ xform
(defun xform-drc (drc-wafer)
  (make-wafer :resist (drc-wafer-resist drc-wafer)
           :metal (drc-wafer-metal drc-wafer)))

(defun xform2 (named-wafer)
  (cons (car named-wafer)
```

```
      (xform-drc (cdr named-wafer))))

(defun xform (wafer-list)
  (mapcar #'xform2 wafer-list))

(defun update-advice-waferstate (task)
  (let ((waferstate (xform *wafer-list*)))
    (set-task-waferstate task waferstate)))




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;; *WAFER-STEPS* that are traced to update the wafer state                  ;;
;; ========================================================                 ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defvar *wafer-steps*

  ;; process predicate                  change wafer op

  '((deposits-resist          .     deposit-resist)
    (etches-resist              .       etch-resist)
    (deposits-metal           .     deposit-metal)))


(defun rule-process-predicate (rule)  (car rule))
(defun rule-wafer-predicate (rule) (cdr rule))

(defun step-predicate (step) (car step))
(defun step-wafer-op (step) (cdr step))


; deposits-resist
;-------------------

(defun deposits-resist (task)
  (let* ((form (task-flow task))
         (chws (change-wafer-state form)))
    (if chws
      (if (listp (first chws))
          (let ((flat-chws (flatten chws)))
            (member t (mapcar #'resist-check-c flat-chws)))
        (resist-check-c chws)))))

;         (dolist (op-part chws)
;               (resist-check op-part))
;       (resist-check chws)))))

(defun resist-check-c (chws)
  (cond ((and (listp chws) (keywordp (first chws)))
         (let ((cws-primitive (first chws)))
           (cond ((eq cws-primitive :deposit)
                  (resist? (fl-form-keyvalue :material chws)))))))))

(defun deposit-resist (wafer)
```

```
  (setf (drc-wafer-resist (cdr wafer)) t)
  (setf *has-seen-photo-step* t))          ; this is for rule#16


(defun resist? (material)
  (cond ((or (eq material :resist)
             (eq material :positive-resist)
             (eq material :negative-resist)
             (eq material :exposed-positive-resist)
             (eq material :exposed-negative-resist)
             (eq material :developed-positive-resist)
             (eq material :developed-negative-resist)
             (eq material :baked-resist)
             (string-equal material "Kodak 820"))
         material)
        (t nil)))


; etches-resist
;----------------

(defun etches-resist (task)
  (let* ((form (task-flow task))
         (chws (change-wafer-state form)))
    (if chws
      (if (listp (first chws))
          (let ((flat-chws (flatten chws)))
            (member t (mapcar #'etch-check flat-chws)))
        (etch-check chws)))))

(defun etch-check (chws)
  (cond ((and (listp chws) (keywordp (first chws)))
         (let ((cws-primitive (first chws)))
           (cond ((eq cws-primitive :etch)
                (resist? (fl-form-keyvalue :material chws)))))))))

(defun etch-resist (wafer)
  (setf (drc-wafer-resist (cdr wafer)) nil))


; deposits-metal
;----------------

(defun deposits-metal (task)
  (let* ((form (task-flow task))
         (chws (change-wafer-state form)))
    (if chws
      (if (listp (first chws))
          (let ((flat-chws (flatten chws)))
            (member t (mapcar #'metal-check flat-chws)))
        (metal-check chws)))))

(defun metal-check (chws)
  (cond ((and (listp chws) (keywordp (first chws)))
         (let ((cws-primitive (first chws)))
           (cond ((eq cws-primitive :deposit)
                (metal? (fl-form-keyvalue :material chws)))))))))

(defun deposit-metal (wafer)
  (setf (drc-wafer-metal (cdr wafer)) t))
```

```
(defun metal? (material)
  (cond ((or (eq material :aluminum)
             (eq material :gold)
             (eq material :silver)
             (string-equal material "metal")
             (string-equal material "aluminum")
             (string-equal material "gold")
             (string-equal material "silver"))
         material)
        (t nil)))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                            ;;
;;              *SAFETY-RULES* that are being checked                         ;;
;; ========================================================================   ;;
;;                                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defvar *safety-rules*

  ;; process step predicate              wafer state predicate THAT
BLOCKS

 '((resist-not-allowed        .     resist-present)))

;    (ever-metallized-not-allowed     .      ever-metallized)
;    (uses-tube                  .     phosphorus-doped-oxide-present)
;    (metal-not-allowed          .     metal-present)
;    (polyimide-not-allowed        .       polyimide-present)

;; waiting for spec from linus
;   (no-hi-phos-allowed           .       hi-conc-phos-present)
;   (uses-thin-ox-tube          .     not-hi-resistivity-si-surface)

;    ((and uses-thin-ox-tube not-thin-ox)   . nil)))



;; predicates that block:
;; ---------------------

(defun resist-present (wafer)
  (drc-wafer-resist (cdr wafer)))

(defun metal-present (wafer)
  (drc-wafer-metal (cdr wafer)))

(defun failed (anything) t)


;;-------------------------------------------------
;; routines for PhotoResist related *safety rule*'s  |
;;-------------------------------------------------


;;
```

```
;; (1) No PR in Diffusion Tubes
;; ============================
;; (resist-not-allowed   .   resist-present)

(defun resist-not-allowed (task)
  (let* ((form (task-flow task))
         (trmt (treatment form)))
    (if trmt
      (if (listp (first trmt))
          (let ((flat-trmt (flatten trmt)))
            (mapcar #'resist-check-t flat-trmt))
        (resist-check-t trmt)))))

(defun resist-check-t (treatment)
  (cond ((and (listp treatment) (keywordp (first treatment)))
         (let ((treatment-primitive (first treatment)))
           (eq treatment-primitive (or :furnace
                                       :thermal))))))

(defun flatten (crap)
  (if (listp crap)
      (if (listp (first crap))
        (apply #'append (mapcar #'flatten crap))
      (list crap))))

(defun flatten2 (crap)
  (if (listp crap)
      (if (listp (first crap))
        (apply #'append (mapcar #'flatten2 crap))
      crap)))


;;
;; (2) No PR in RCA Clean
;; ======================
;; (no-resist-in-RCA     .    resist-present)

(defun no-resist-in-RCA (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'rca? mach)))
    (if (member t result) t)))

(defun rca? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "rca") t)))


;;
;; (3) No PR in Nitride Wet Etch
;; =============================
;; (no-resist-in-Nitride-Wet-Etch . resist-present)

(defun no-resist-in-Nitride-Wet-Etch (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'nitride? mach)))
    (if (member t result) t)))

(defun nitride? (machine)
  (let ((name (machine-name machine)))
    (if (string-eqtaldeame "nitride") t)))
```

```
;;
;; (4) No PR in Varian Metallization System
;; ========================================
;; (no-resist-in-Varian    .    resist-present)

(defun no-resist-in-Varian (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'varian? mach)))
    (if (member t result) t)))

(defun varian? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "varian") t)))



;;-----------------------------------------------------
;; routines for Metal related *safety rule*'s  |
;;-----------------------------------------------------

;;
;; (5) No Metal in RCA Clean
;; =========================
;; (no-metal-in-RCA       .      metal-present)

(defun no-metal-in-RCA (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'rca? mach)))    ;rca? defined in rule#2
    (if (member t result) t)))


;;
;; (6) No Metal in Piranha Clean
;; =============================
;; (no-metal-in-piranha    .    metal-present)

(defun no-metal-in-piranha (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'piranha? mach)))
    (if (member t result) t)))

(defun piranha? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "pre-metal") t)))

;;
;; (7) No Metal in Oxide Etch
;; ==========================
;; (no-metal-in-oxide     .      metal-present)

(defun no-metal-in-oxide (task)
  (let* ((mach (task-machines task))
         (name (task-name task))
         (result (mapcar #'oxide? mach)))
    (if (and (member t result)
             (not (string-equal name "FINAL-RINSE")))
      t)))
```

```
(defun oxide? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "oxide") t)))


;;
;; (8) No Metal in Nitride Wet Etch
;; ================================
;; (no-metal-in-nitride    .     metal-present)

(defun no-metal-in-nitride (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'nitride? mach)))
    (if (member t result) t)))

(defun nitride? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "nitride") t)))

;;
;; (9) Metal only allowed in tubes B7, B8  (= No Metal in /tubeB7,
/tubeB8)
;;
=============================================================================
;; (no-metal-in-tubes     .       metal-present)

(defun no-metal-in-tubes (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'tubes? mach)))
    (if (member t result) t)))

(defun tubes? (machine)
  (let ((name (machine-name machine)))
    (if (and (typep machine 'furnace)
             (not (or (string-equal name "tubeB7")
                      (string-equal name "tubeB8"))))
      t)))



;;
;; (10) No Quartz Wafers in Varian Metallization System
;; ====================================================
;;       (no-quartz-in-varian  .  failed)

(defun no-quartz-in-Varian (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'varian? mach)))     ;varian? defined in (4)
    (if (member t result)
      (check-for-quartz task))))

(defun check-for-quartz (task)
  (let* ((wsets (task-wafersets task))
         (result (mapcar #'check2 wsets)))
    (if (member "quartz" result :test #'string-equal) "quartz")))

(defun check2 (wset)
  (let ((wafers (waferset-wafers wset)))
    (if (member "quartz" (mapcar #'check3 wafers) :test #'string-equal)
      "quartz" )))
```

```
(defun check3 (wafer)
  (let* ((type (wafer-type wafer))
         (name (wafertype-name type)))
    name))



;;-----------------------------------------------------
;; routines for Sequence related *safety rule*'s  |
;;-----------------------------------------------------
;;
;; these rules are independent of wafer state
;;

;;
;; (11) Must have oxide etch prior to nitride wet etch
;; ===================================================
;    (need-oxide-before-nitride   .   failed)

(defun need-oxide-before-nitride (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'nitride? mach)))    ;nitride defined in (8)
    (if (member t result)
      (failed-oxide-before-nitride task))))

(defun failed-oxide-before-nitride (task)
  (let* ((prev1 (first (task-prev_leaf_tasks task)))
         (prev2 (if prev1
                    (first (task-prev_leaf_tasks prev1))
                  nil))
         (mach-p1 (if prev1 (task-machines prev1) nil))
         (mach-p2 (if prev2 (task-machines prev2) nil)))
    (if (not (or (member t (mapcar #'oxide? mach-p1))     ;oxide defined
in (7)
                 (and (member t (mapcar #'oxide? mach-p2))
                      (member t (mapcar #'inspection? mach-p1)))))
      t)))

(defun inspection? (machine)
  (let ((name (machine-name machine)))
    (if (or (string-equal name "ellipsometer")
            (string-equal name "nanospec"))
      t)))


;;
;; (12) Wafers must go through Coater prior to Stepper
;; ===================================================
;;     (must-have-coater-before-stepper   .   failed)

(defun must-have-coater-before-stepper (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'stepper? mach)))
    (if (member t result)
      (failed-coater-before-stepper task))))

(defun stepper? (machine)
  (let ((name (machine-name machine)))
```

```
    (if (string-equal name "stepper") t)))


(defun failed-coater-before-stepper (task)
  (let* ((prev1 (first (task-prev_leaf_tasks task)))
       (prev2 (if prev1 (first (task-prev_leaf_tasks prev1)) nil))
       (mach-p1 (if prev1 (task-machines prev1) nil))
       (mach-p2 (if prev2 (task-machines prev2) nil)))
    (if (not (or (member t (mapcar #'coater? mach-p1))
             (and (member t (mapcar #'coater? mach-p2))
                  (member t (mapcar #'inspection? mach-p1)))))
      t)))

(defun coater? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "coater") t)))


;;
;; (13) Wafers must go through Developer following Stepper
;; ======================================================
;;    (must-have-developer-after-stepper  .  failed)

(defun must-have-developer-after-stepper (task)
  (let* ((mach (task-machines task))
       (result (mapcar #'stepper? mach)))    ;stepper defined in (12)
    (if (member t result)
      (failed-coater-before-stepper task))))

(defun failed-coater-before-stepper (task)
  (let* ((next1 (first (task-next_leaf_tasks task)))
       (next2 (if next1 (first (task-next_leaf_tasks next1)) nil))
       (mach-n1 (if next1 (task-machines next1) nil))
       (mach-n2 (if next2 (task-machines next2) nil)))
    (if (not (or (member t (mapcar #'developer? mach-n1))
             (and (member t (mapcar #'developer? mach-n2))
                  (member t (mapcar #'inspection? mach-n1)))))
      t)))

(defun developer? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "developer") t)))


;;
;; (14) BPSG deposition must be followed by BPSG flow
;; =================================================
;;   (must-have-bpsg-flow-after-dep   .   failed)

(defun must-have-bpsg-flow-after-dep (task)
  (let* ((mach (task-machines task))
       (result (mapcar #'BPSG? mach))
       (super (task-supertask task))
       (super-name (if super (task-name super) nil)))
    (if (and (member t result)
           (not (string-equal super-name "LTO-DEPOSITION")))
      (failed-bpsg task))))

(defun BPSG? (machine)
  (let ((name (machine-name machine)))
```

```
    (if (string-equal name "tubeA8") t)))

(defun failed-bpsg(task)
  (let* ((next1 (first (task-next_leaf_tasks task)))
       (next2 (if next1 (first (task-next_leaf_tasks next1)) nil))
       (mach-n1 (if next1 (task-machines next1) nil))
       (mach-n2 (if next2 (task-machines next2) nil)))
    (if (not (or (member t (mapcar #'BPSG-flow? mach-n1))
              (and (member t (mapcar #'BPSG-flow? mach-n2))
                   (member t (mapcar #'inspection? mach-n1)))))
      t)))

(defun BPSG-flow? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "tubeB6") t)))



;;
;; (15) Must have oxide etch following Phos. Dep
;; ============================================
;;    (must-have-oxide-after-phos  .  failed)

(defun must-have-oxide-after-phos (task)
  (let* ((mach (task-machines task))
       (result (mapcar #'Phos-dep? mach)))
    (if (member t result)
      (failed-oxide-before-phos task))))

(defun Phos-dep? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "tubeA4") t)))

(defun failed-oxide-before-phos (task)
  (let* ((next1 (first (task-next_leaf_tasks task)))
       (next2 (if next1 (first (task-next_leaf_tasks next1)) nil))
       (mach-n1 (if next1 (task-machines next1) nil))
       (mach-n2 (if next2 (task-machines next2) nil)))
    (if (not (or (member t (mapcar #'oxide? mach-n1))
              (and (member t (mapcar #'oxide? mach-n2))
                   (member t (mapcar #'inspection? mach-n1)))))
      t)))



;;
;; (16) Wafers must have pre-metal clean before metallization if ever
;;      preceeded by a photo step
;; ================================================================
;;     (must-have-pre-metal-clean   .  failed)

(defun must-have-pre-metal-clean (task)
  (if (and (deposits-metal task)
         *has-seen-photo-step*)
      (no-pre-metal-clean task)))

(defun no-pre-metal-clean (task)
  (let* ((prev1 (first (task-prev_leaf_tasks task)))
       (prev2 (first (task-prev_leaf_tasks prev1)))
       (mach-p1 (task-machines prev1))
       (mach-p2 (task-machines prev2)))
```

```
    (if (not (or (member t (mapcar #'pre-metal? mach-p1))
                 (and (member t (mapcar #'pre-metal? mach-p2))
                      (member t (mapcar #'inspection? mach-p1)))))
      t)))

(defun pre-metal? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "pre-metal") t)))



;;
;; (17a) Only high-resistivity (> 1ohm/cm) Si in tube A2
;;=======================================================
;;    (only-high-resistivity-in-a2   .   failed)

(defun only-high-resistivity-in-a2 (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'tubeA2? mach)))
    (if (member t result)
      (check-resistivity task))))

(defun check-resistivity (task)
  (let* ((wsets (task-wafersets task))
         (result (mapcar #'check-res2 wsets)))
    (if (member t result) t )))

(defun check-res2 (wset)
  (let ((wafers (waferset-wafers wset)))
    (if (member t (mapcar #'check-res3 wafers)) t)))

(defun check-res3 (wafer)
  (let* ((type (wafer-type wafer))
         (epi-res (wafertype-epi_resistivity type))
         (sub-res (wafertype-substrate_resistivity type))
         (resistivity (if epi-res epi-res sub-res))
         (result (floatinterval-lower resistivity)))
    (if (< result 1) t)))


;;
;; (17b) Only thin oxide growth in tube A2
;;=========================================
;;    (only-thin-ox-in-a2   .   failed)

(defun only-thin-ox-in-a2 (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'tubeA2? mach)))
    (if (member t result)
      (check-oxide-thickness task))))

(defun check-oxide-thickness (task)
  (let* ((form (task-flow task))
         (chws (change-wafer-state form)))
    (if chws
      (if (listp (first chws))
          (let ((flat-chws (flatten chws)))
            (member t (mapcar #'cot2 flat-chws)))
        (cot2 chws)))))
```

```
(defun cot2 (chws)
  (cond ((and (listp chws) (keywordp (first chws)))
         (let ((thick (fl-form-keyvalue :thickness chws)))
           (if thick
               (let ((thick2 (inexact-mean thick)))
               (> thick2 500)))))))

(defun tubeA2? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "tubeA2") t)))


;;
;; (18) Wafers should not go directly into tube A1 or A2 after Phos.
;; ================================================================
;;    (cannot-enter-tube-A1-or-A2-after-Phos  .  failed)

(defun cannot-enter-tube-A1-or-A2-after-Phos (task)
  (let* ((mach (task-machines task))
         (result (mapcar #'Phos-dep? mach)))
    (if (member t result)
      (failed-A1-or-A2 task))))

(defun Phos-dep? (machine)
  (let ((name (machine-name machine)))
    (if (string-equal name "tubeA4") t)))

(defun failed-A1-or-A2 (task)
  (let* ((next1 (first (task-next_leaf_tasks task)))
         (next2 (if next1 (first (task-next_leaf_tasks next1)) nil))
         (next3 (if next2 (first (task-next_leaf_tasks next2)) nil))
         (mach-n1 (if next1 (task-machines next1) nil))
         (mach-n2 (if next2 (task-machines next2) nil))
         (mach-n3 (if next3 (task-machines next3) nil)))
    (if (or (member t (mapcar #'A1-or-A2? mach-n1))
            (member t (mapcar #'A1-or-A2? mach-n2))
            (member t (mapcar #'A1-or-A2? mach-n3)))
      t)))

(defun A1-or-A2? (machine)
  (let* ((name (machine-name machine)))
    (if (or (string-equal name "tubeA1")
            (string-equal name "tubeA2"))
      t)))


;;-----------------------------------------------------------------------

(setf *safety-rules*
      '((resist-not-allowed            . resist-present)      ; (1)
      (no-resist-in-RCA               . resist-present)       ; (2)
      (no-resist-in-Nitride-Wet-Etch . resist-present)        ; (3)
      (no-resist-in-Varian            . resist-present)       ; (4)
      (no-metal-in-RCA                . metal-present)        ; (5)
      (no-metal-in-piranha            . metal-present)        ; (6)
      (no-metal-in-oxide              . metal-present)        ; (7)
      (no-metal-in-nitride            . metal-present)        ; (8)
      (no-metal-in-tubes              . metal-present)        ; (9)
      (need-oxide-before-nitride      . failed)               ; (11)
      (must-have-coater-before-stepper . failed)              ; (12)
```

```
      (must-have-developer-after-stepper . failed)            ; (13)
      (must-have-bpsg-flow-after-dep   .   failed)            ; (14)
      (must-have-oxide-after-phos  .  failed)                 ; (15)
      (cannot-enter-tube-A1-or-A2-after-Phos  .  failed)      ; (18)
      (must-have-pre-metal-clean   .  failed)                 ; (16)
      (only-high-resistivity-in-a2   .   failed)              ; (17a)
      (only-thin-ox-in-a2   .   failed)                       ; (17b)
      (no-quartz-in-varian  .  failed)))                      ; (10)
```

BIBLIOGRAPHY

[1]Wolf, S., Tauber, R., *Silicon Processing for the VLSI Era, Volume I - Process Technology*, Lattice Press, 1986. pp. xxiii.

[2] McIlrath, M., Troxel, D., Heytens, M., Penfield, P., Boning, D., Jayavant, R., "CAFE - The MIT Computer-Aided Fabrication Environment," *IEEE Transactions on Components, Hybrids and Manufacturing Technology*, vol. 15, no. 3, pp. 353-360, June 1992.

[3] McIlrath, M., Boning, D., "Integrating Semiconductor Process Design and Manufacture Using a Unified Process Flow Representation," in *Proc. Second Int. Conf. on Computer Integrated Manufacturing*, Troy, NY, May 1990, pp. 224-230.

[4] Fischer, G., "Beginners Guide to Fabrication Using CAFE," *MIT CIDM Memo no. 92-3*, Oct. 1992

[5] Baird, H. "Fast Algorithms for LSI Artwork Analysis,". in *Proceedings of the 14th Design Automation Conference*, pp. 303-311. June 1977.

[6] Seiler, L., *A Hardware Assisted Methodology for VLSI Design Rule Checking,* Ph.D. thesis, Massachusetts Institute of Technology, February 1985.

[7] Blank, T., Stefik, M., vanCleemput, W. "A Parallel Bit Map Processor Architecture for DA Algorithms," in *Proceedings of the 18th Design Automation Conference*, pp. 837-845, June 1981.

[8] Longhead, R., McCubbrey, D., "The Cytocomputer: A Practical Pipelined Image Processor," in *Proceedings of the 7th Annual International Symposium on Computer Architecture*, pp. 271-277, May 1980.

[9] Wittenmeyr, D., *Offline Design Rule Checking for VLSI*, S.M. thesis, University of Toledo, 1992.

[10] Wenstrand, J., *An Object-Oriented Model for Specification, Simulation, and Design of Semiconductor Fabrication Processes*, Ph.D. thesis, Stanford University, March 1991.

[11] Steele Jr., G., *The Definition and Implementation of a Computer Programming Language Based on Constraints*, Ph.D. thesis, Massachusetts Institute of Technology, August 1980.

[12] Balemi, S., Hoffmann, G., Gyugyi, P., Wong-Toi, H, Franklin, G., "Supervisory Control of a Rapid Thermal Multiprocessor," *Joint Automatica - IEEE Transaction on Automatic Control Special Issue on "Meeting the Challenge of Computer Science in Industrial Applications of Control"*, Nov. 5, 1991.

[13] Funakoshi, K., Mizuno, K., "A Rule Based VLSI Process Flow Validation System With Macroscopic Process Simulation," in *IEEE Transactions on Semiconductor Manufacturing*, vol. 3, no. 4. Nov. 1990.

[14] Heytens, M., Nikhil, R., "GESTALT: An Expressive Database Programming System," in *ACM SIGMOD Rec.*, vol 18, no. 1, pp. 54-67, Mar. 1989

[15] Steele Jr., G., *Common Lisp: The Language*, 2nd ed., Bedford, MA: Digital, 1990, pp. 770-864.

[16] Boning, D., McIlrath, M., "Guide to the Process Flow Representation", *MIT MTL Memo no. 93-724*, MIT Microsystems Res. Ctr. September 1993.

[17] McIlrath, M., "An Automata-Based Approach to Process Rules," *Unpublished Memo*, Oct. 1993.

[18] Woo, A., *A Generic Graphical Tree Editor for Wafer Processing*, S.M. thesis, Massachusetts Institute of Technology, May 1993.