SEMICONDUCTOR PROCESS DESIGN: REPRESENTATIONS, TOOLS, AND METHODOLOGIES

by

Duane S. Boning

S.M., Massachusetts Institute of Technology (1986)

submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment

of the requirements for the degree of

DOCTOR OF PHILOSOPHY

IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 4, 1991

Signature of Author _____

Department of Electrical Engineering and Computer Science

January 4, 1991

Certified by _____

Dimitri A. Antoniadis

~~Thesis~~ Supervisor

Accepted by _____

Arthur C. Smith

Chairman, Departmental Committee on Graduate Students

Semiconductor Process Design: Representations, Tools, and Methodologies

Duane S. Boning

Submitted to the Department of Electrical Engineering and Computer Science on January 4, 1991 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

A vision of the future is presented in which the design of semiconductor fabrication processes is an integral part of the design of application specific circuits. The circuit designer (particularly of analog and microelectromechanical devices and circuits) will no longer be restricted to processing specification solely via layout and mask information, but will also directly specify process parameters, step sequences, and whole fabrication processes. For this vision to be realized, new approaches and software support systems are needed for both semiconductor process design and fabrication.

This thesis focuses on the problem of semiconductor process design within the context of a comprehensive semiconductor CAD/CIM (computer-aided-design/computer-aided manufacturing) approach. Three essential ingredients for the advancement of process design are here described. First, *representations* of the designed artifacts, specifically structures to be fabricated and fabrication processes, have been developed and adopted. This work contributes a prototype PIF (Profile Interchange Format) database to facilitate the uniform representation of wafer structures. A process flow representation (PFR) provides a representation of the process suitable for both design uses and fabrication. Based on these representations, new *tools* providing CAD capabilities beyond that of process simulation alone have been prototyped. A Simulation Manager provides uniform interfaces between the process flow and process simulators (Suprem-III and Simpl-2). Additional tools have been prototyped, including Process Advisors to provide help in process synthesis. Finally, *methodologies* for process design have been investigated, and the concept of "mutators" is introduced to aid in process integration. These representations, tools, and methodologies contribute toward the advancement of CAD/CIM systems that will be necessary to support the design and execution of application specific processes.

Thesis Supervisor: Dimitri A. Antoniadis
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis has been made possible only through the help and guidance of many people, and parts of this thesis are the results of collaborative work. While these contributions will be specifically cited in this thesis, I would like to take the opportunity to thank these individuals here.

The generic semiconductor process model owes its origin to Paul Penfield, and its evolution has been influenced by many discussions with Paul Penfield, Michael McIlrath, Emmanuel Sachs, and Robert Harris.

I would like to acknowledge the early contributions of Thye-Lai Tung in work on storage mechanisms and program interfaces for the Profile Interchange Format (PIF). Several undergraduate students were involved with the PIF at various stages in its evolution, including Ronald Duncan, Deniz Akkus, David Hamilton, and Partha Saha. Michael Heytens has been instrumental in providing database and interface mechanisms so that the PIF could be layered on Gestalt. I thank Alexander Wong of UC Berkeley for collaboration in our effort to develop and "standardize" on PIF program interfaces. I also thank all of the members of standards groups related to the PIF, including the EDIF Technical Subcommittee on Process/Device Representation (PIF/EDIF), and the TCAD Framework Semiconductor Wafer Representation Working Group.

The Process Flow Representation (PFR) owes its vision to Paul Penfield and Michael McIlrath. Initial implementations of the PFR were done by Michael McIlrath, who generously assisted me in evolving the PFR for use in simulation and process design. I send Mike good wishes as he continues his efforts to make the PFR work equally well for fabrication.

This thesis has been performed under the umbrella of the MIT Computer Aided Fabrication (CAF) project guided by Paul Penfield and Dimitri Antoniadis. The core of the CAF project has been the development of CAFE, the facility support

system. This thesis would not have been possible without the efforts of Donald Troxel and his CAFE implementation team including Mike McIlrath, Mike Heytens, Rajeev Jayavant, Will Martinez, Abbas Kashani, and a host of others.

Many thanks to Prabha Tedrow for her patience in transmitting a portion of her vast knowledge of MIT's CMOS baseline process to me. Further thanks to Nestore Polce for assistance in modifications and updates to the PFR description of the baseline process.

I owe a very sincere debt of gratitude to Dimitri Antoniadis, who has been my mentor and advisor in the most true sense.

Many, many thanks to fellow students Jarvis Jacobs and Robert Harris for what has been many years of shared intellectual endeavor and supportive friendship. My appreciation goes out to the other students, staff, and faculty I have been associated with at MIT.

I thank my wife Peggy for her support, her energy, and her love. I dedicate this thesis to Peggy and to our child, William Cynric Boning, born October 10, 1990.

This work has been performed in the MIT Microsystems Technology Laboratories as part of the Computer Aided Fabrication (CAF) research project. Support for this research has been provided in part by DARPA contract #MDA-972-88-K-0008 and by Texas Instruments, Inc. Support for my graduate work has also been provided by the National Science Foundation through a Graduate Fellowship, and by Intel Corp. through an Intel Graduate Fellowship.

Biography

Duane Boning earned the Bachelor of Science in Electrical Engineering and the Bachelor of Science in Computer Science degrees in 1984, and the Master of Science in Electrical Engineering and Computer Science in 1986, all at the Massachusetts Institute of Technology. He is a member of the Tau Beta Pi, Eta Kappa Nu, and Sigma Xi honorary societies, and is a member of the IEEE and the ACM. He has been a General Motors Scholar, a National Science Foundation Graduate Fellow, and an Intel Graduate Fellow. He has served as Chairman of the EDIF (Electronic Design Interchange Format) technical subcommittee on Process/Device, and as Chairman of the Semiconductor Process Representation (SPR) working group of the TCAD Framework Group within the CAD Framework Initiative (CFI). His research interests are in the computer aided design of semiconductor fabrication processes and devices, and in the computer integrated manufacturing of integrated circuits.

Contents

Chapter 1

Introduction

This thesis is concerned with semiconductor fabrication process design. The need to design and enhance fabrication processes has and will continue to grow in the future as new circuit, device, and time demands are placed on process development. This thesis focuses on the fundamental changes that must be made, philosophically and technically, to take the greatest advantage of semiconductor processing in the future.

Process and device (or "technology") CAD must evolve substantially to better support semiconductor process design. This thesis draws on the experience and conclusions of earlier work with MASTIF (MIT Analysis and Synthesis Tool for IC Fabrication) [1, 2, 3]. That work emphasized the analogy between technology CAD (TCAD) and electronic CAD (ECAD), and focused on the integration of tools and the improvement of user interfaces to TCAD tools. It was found that these are necessary but are not sufficient to provide a powerful and extensible semiconductor process design capability.

Process design capability must also grow in three additional ways. First, the design of semiconductor technologies must focus more closely on the objects being designed with attention to both product and process design. A mental shift to the *representations* that the designer must generate and manipulate is necessary. Second, the *tools* that support the act of technology design require substantial growth, and

new kinds of CAD tools are needed. Currently, technology design is "tool" driven: one focuses on the simulation and characterization tools available. A focus on the representations clarifies the design activity, pinpoints the kinds of tools that are necessary, and greatly influences the construction of the TCAD tools themselves. Finally, this thesis considers the possibility of entirely new technology design *methodologies*. Not only must one make the tool-centric to data-centric transition, but one must also think about the problem and opportunities of process design in a new way.

The central goal of this thesis is to contribute toward better semiconductor process *design* capability. Enabling better integration of process design and manufacturing activities is also a goal of this work. The guiding vision behind these goals is the radical notion that *application specific processing* is possible and may well open up entirely new and unforeseen innovation at the circuit design level. In the future, implementation of a circuit will include specifications of the product and process that go far beyond the mask set.

In Part I, the opportunity for and central vision of application specific process design is discussed. Having done so, the rest of this thesis will discuss the software components and systems that must be put in place to support that vision. Considering such requirements from the ground up, the understanding of the basic building blocks of the design is crucial. Integrated circuit fabrication fundamentally consists of small manufacturing steps performed on wafers. The fundamental model of semiconductor fabrication used in this work is discussed in Chapter 4.

Part II of this thesis focuses on the basic *representation* of processes and wafers. Chapter 5 discusses the "Process Flow Representation" or PFR used in this work. Chapter 6 discusses issues involving the representation of semiconductor wafers for use in process simulation, specification, and other aspects of design and fabrication.

Once the basic representations of the design objects are in place, it becomes possible to view CAD *tools* as performing or aiding in the manipulation and transformation of these representations. The role of technology CAD tools in general, and

more specifically investigations of tools to support process simulation and process synthesis, are discussed in Part III.

The representations of the wafer and the process, coupled with tools to manipulate these representations, are themselves simply building blocks that must be used in an intelligent fashion to design, analyze, debug, and otherwise create or investigate semiconductor processes. The *methodologies* that are needed to guide and direct the use of tools and the manipulation of data are discussed in Part IV. These representations, tools, and methodologies are demonstrated through the description and analysis of the MIT CMOS baseline process. Finally, an enhancement of the baseline process to include a capacitor with specified electrical characteristics (voltage sensitivity and unit area capacitance) further demonstrates the approaches described in this thesis for the support of application specific processing.

Part I

The Vision: Application Specific Processing

Chapter 2

A Process Design and Manufacture (R)Evolution

The analogy between VLSI design and semiconductor process design is a strong one. Both are large, complicated design tasks that demand computer aids. The two differ in the degree of maturity in these computer aids. A revolution in process design comparable to the VLSI revolution of the 1980's requires many evolutionary advances in these computer aided design capabilities. The analogy provides further guidance in the evolutionary and revolutionary changes that are necessary in both design and execution to achieve better process design, and ultimately to achieve *application specific processing*. Drawing on the lessons of the VLSI revolution, these requirements (in both the VLSI and process design cases) can be depicted in the graph of Figure 2.1. If one focuses on the *fabrication* of a process, then a foundry or implementation service is needed, which requires both manufacturing constraints or design rules (defining the capability of the foundry), and representations of the design to be manufactured (the "work order" sent to the foundry). If, on the other hand, one is essentially concerned with the requirements for efficient and capable process *design*, then methodologies,

†The notion of *application specific processing* which underlies and motivates the MIT CAF system as well as the work of this thesis originated with Paul Penfield.

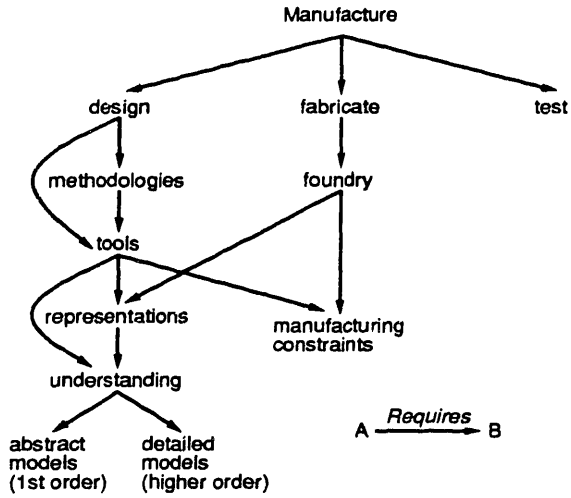


Figure 2.1: ASIC/ASP revolution/evolution requirements.

tools, and representations become essential. A revolution in the manufacture of innovative devices and structures will require improvements in the design, fabrication, and test of semiconductor processes.

This chapter considers three aspects of *application specific processing*. First, the opportunity for and desirability of increased flexibility in the specification and execution of semiconductor processes is discussed. Second, the impact of application specific processes on *fabrication* is considered. Finally, the analogy to VLSI design is examined to set the stage for the necessary changes in process *design* toward which this thesis contributes.

2.1 The Promise of Application Specific Processing

Custom processes might be described as any new process designed to support a new family of circuits. Most process design that occurs today fits this description, and typically requires on the order of two years for development and transfer to manufac-

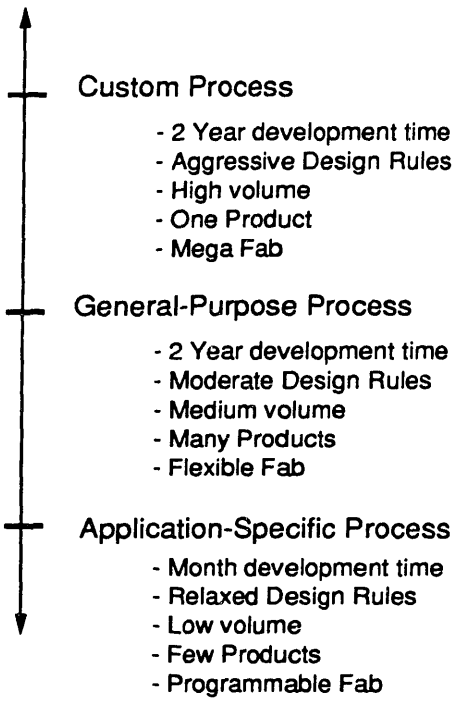


Figure 2.2: Spectrum of process types from aggressive, general purpose *custom* processes to *application specific processes*.

turing. An *Application specific process* (ASP), on the other hand, can be defined as a process that is modified or implemented differently for each application (and thus circuit) specific chip. There is a spectrum, illustrated in Figure 2.2, from custom to fully application specific processes, depending on the length of time it takes to implement the process, and more directly on the volume of product that the process applies to. Highly tuned or customized processes include those for high volume chips such as memories; general-purpose processes include CMOS and BiCMOS processes intended to support large families of digital or mixed digital-analog circuits; and application specific processes might include microelectromechanical processes integrating sensors on a chip [4].

A critical question must be asked: is there a need for application specific processes? Certainly there is already a large demand for custom and flexible processes.

What are the benefits and costs associated with application specific processing, and what demand might one expect to emerge for application specific processes and programmable fabrication?

The answer hinges on the need for process-level modifications in order to achieve device and circuit-level, and thus system level requirements. Currently, there is a pyramid structure that justifies the high cost of process development. Either huge numbers of commodity chips (memories or microprocessors) are produced using a custom process, or huge numbers of circuits using the same transistor or device family are designed with only mask-level changes on top of a given process. Furthermore, most VLSI circuits are intentionally designed in a "technology independent" fashion, so that the same circuit can be fabricated by different vendors, or so that the circuit can expect to take advantage of iterative cost, chip-size, or performance improvements in a process and its corresponding devices. Application specific processes run precisely counter to the independence between circuits and processes usually striven for. What kinds of circuits now or in the future might require or justify process specificity? Are there subsets of circuit designs that can (1) be made better, or (2) be made possible if one could make changes to the underlying processes? The interface between circuit and process lies at the device level; is there a need for tailored devices in circuit design? The answer is is yes, but more so for certain subsets of circuit design.

Digital processes

It does not appear that there is a great deal of need or opportunity for custom processing in digital circuit design. One reason for the success of the digital approach is the imposition of the "digital abstraction" which purposefully insulates circuit designers from the details of the underlying devices. For digital design, then, the basic components are fixed, and device or process level flexibility is undesirable.

An important peripheral impact of application specific process design capability should be significantly reduced barriers to the prototyping and development of new

digital processes. It is not expected, however, that particular designs or even families of digital designs will require direct access to processing flexibility.

Analog processes

The current practice in analog process design is limited negotiation and communication between the circuit designers, the device/process designers, and fabrication groups. The coupling among the individuals or organizations performing these functions is usually much closer than in digital design, suggesting that further integration of these activities would be a major benefit. The goal of application specific processing is to break down these barriers, ultimately to the point where circuit designers are empowered to directly rather than indirectly specify the fabrication of unique devices.

Following are examples of the kinds of process flexibility that analog designers might take advantage of:

- The availability of a set of transistors with a setable threshold voltage (within some range).
- A capacitor with a setable oxide thickness (a controllable capacitance per area).
- In a two-layer polysilicon process, control over the resistivity of one of those layers.
- Control over the temperature coefficient of a polysilicon resistor.

Some examples of application specific processes that have been necessary or requested within the Microsystems Technologies Laboratories at MIT include [5]:

- A high-speed BiCMOS process for mixed analog/digital applications (for a pipelines A/D converter).
- Development of a CCD/CMOS process for vision processing applications.
- Process enhancements to achieve a low capacitor voltage coefficient for an operational amplifier design.
- The investigation of switched capacitor networks as an alternative to resistor grids in analog image processing.
- Process enhancements to provide both normal and p/n depletion transistors.

- Development of a low voltage, very low resistance power MOSFET for synchronous rectification.
- The merge of NMOS double-diffused transistors (DMOS) with conventional CMOS transistors for high performance analog circuits.
- The development of NMOS transistors with ion beam implanted channel regions (where implants are tailored for individual transistors).

Such examples show different degrees of application specific fabrication: (1) parameterization of the basic devices a process makes available; (2) inclusion of substantial amounts of additional processing, and (3) control over details of the process to trade off various device-circuit design aspects. These kinds of process enhancements are often necessary in order to support innovations in novel structures, devices, *and* circuits.

Microelectromechanical processes

Just as in conventional processes, the design and implementation of processes for the construction of micromechanical or microelectromechanical requires well-characterized and well-developed modular processes. The goal of this processing is to construct micromechanical "circuits;" such circuits may require more than simple two-dimensional connectivity of devices. Thus, information in addition to layouts will be necessary for the construction of such circuits.

Application specific processing will, ultimately, require breaking free of the limiting constraints of two-dimensional (lateral) geometry specification. The additional specification that application specific processing will be called on to support includes three dimensional specification (i.e, thicknesses of layers), material mechanical properties of layers (*e.g.*, stiffness of a beam), as well as material or device component electrical properties (the sheet resistivity of a layer or the threshold voltage of an MOS device), and finally combined microelectromechanical properties (the temperature coefficients demanded of a given poly layer).

Micromechanical processes under investigation within the Microsystems Technologies Laboratories at MIT include [5]:

- The integration of capacitive diaphragm sensors with a CMOS process.
- Development of a silicon floating shear-stress piezoresistive sensor.
- Process development for construction of silicon microaccelerometers.
- Investigation of a microvalve for hydraulic system applications.
- Development of several electrostatic microactuators (motors).

The ability to specify and experiment with such processes dramatically opens up the device design space, and thus potentially increases the space of possible circuit and system designs as well. Currently, the design space is segmented and segregated: circuit designers work in one space and device/process designers in another, so that the opportunity for innovation is limited. The potential effect of application specific processing is to throw wide open the doors between circuit and process/device design activities so that these two communities can more closely cooperate and innovate together.

2.2 Fabrication support for ASP

The production of circuits with application specific processes clearly requires the ability to both design and fabricate such processes. The *fabrication* support currently available will have to evolve substantially in order to support application specific processing. The impact of ASPs on fabrication facilities, as summarized in Figure 2.2, is described below.

Megafabs

Megafabs are fabrication facilities that are highly tuned to the low cost, high-volume production of commodity semiconductors (such as DRAMs and microprocessors). The emphasis of computer integrated manufacturing systems in such facilities is on

the tracking of product, the collection of data to support diagnosis and correction of process, and the material transport of product. Yield maximization is of paramount concern.

Flexible Facility

A *flexible facility* is capable of rapidly retooling to run different products and different processes. The number of unique processes is typically much less than the number of unique products being run through the facility. Because multiple product families and processes may be running simultaneously in the same facility, computer integrated manufacturing systems for flexible fabrication must provide enhanced support for control, scheduling, tracking, data collection, and analysis. Yield maximization remains a key concern; minimization of turn-around time is of increasing importance.

Programmable Factories

The execution of an application specific process requires a factory capable of accepting unique designs at both the circuit (layout) and process levels, and performing the necessary manufacturing. CIM systems to support fully programmable factories are also concerned with automation of equipment and equipment communication. Priorities for application specific process production are (1) working silicon on first pass, (2) minimal turn-around time, and (3) maximum yield.

With the substantial demands of application specific processing on flexible facility and CIM system design, a number of research groups are investigating programmable factories, including MIT [6], UC Berkeley, and Stanford [7]. The MIT effort, of which this thesis is a part, is described further in Chapter 3.

2.3 Process Implementation Services

In order to achieve the promise of application specific processing, a fabrication capability must be made available to a larger audience. Just as MOSIS provided a circuit implementation service to the VLSI community, so too will a process implementation service be necessary. Limited process-related services are already emerging and must evolve further.

Unit Process Implementation Services

An interesting recent development is the offering of unit process implementation services by some of the large equipment vendors. For example, in addition to basic diffusion consulting, some customers of BTU Bruce Furnace Corp. request recipes for producing deposited and diffused layers with a given sheet resistance [8]. In order to provide such a service, application engineers must be able to adequately model and simulate the process so as to minimize on-line experimental development of the recipe. Because equipment vendors are experts on the possibilities and limitations of their equipment, and because equipment vendors serve as a natural repository for such implementation expertise, such services may become more common in the future.

Unit Process Foundries

In addition to providing recipe information, a number of unit process fabrication services are available. In addition to suppliers of starting wafers that may include epitaxial silicon layers of desired type, resistivity, and thickness, vendors of mask-making and ion implantation services are becoming commonplace. In many of these cases, the specification of the process goes beyond the masks (*e.g.*, to include implantation energies, doses, etc.). Such services are attractive particularly when the equipment costs are large and in-house use of the equipment does not justify purchase of the machine. As the cost of fabrication equipment increases, the use of such

services in low-volume chip production (particularly by research organizations) may also increase.

Complete Process Implementation Services

These trends suggest that a service that can produce chips given specifications of an entire process is an exciting possibility. Many silicon foundries now exist, where the only process flexibility is that indicated by layout information. MOSIS is a particularly well known example of a clearing-house to provide custom chip-making services. Process flexibility that can be provided with relative ease might include specification of treatment information, such as ion implantation parameters. Over time, the specification of process steps might be at a higher level, where one indicates the design goals and constraints (*e.g.*, polysilicon layer thickness and resistivity), and the service itself implements (designs and executes) a unit process step that meets those requirements.

In order to enable research into novel device structures by research groups without their own fabrication facilities, something like an expanded MOSIS is needed. The availability of a PROCIS, or "Process Implementation Service," open to the research community might enable a similar participatory explosion in device (both electronic and mechanical) innovation.

2.4 VLSI Design (R)Evolution

The availability of a silicon foundry service in the form of MOSIS has helped to make possible widespread VLSI experimentation and education. One can design a "custom chip" by complete specification of the masks that are to be used with a given fabrication process, and remote *fabrication* of a circuit design is possible. This was necessary to enable the VLSI revolution, but was not by itself sufficient. In particular, several *design* capability requirements had to be satisfied as well. In order to guide the

evolution of application specific process technology, it is valuable to review the other ingredients contributing toward the phenomenal growth of VLSI design capability.

Interface to Fabrication

First, a basic bidirectional interface between design and manufacturing was required. In so far as different processes produce devices with similar characteristics, and provide similar design rules, the processes can be considered "compatible" with each other, and the same layout (perhaps in a "technology independent" form) can be used to produce the custom chip. The design rules supported by a facility, then, specify the interface from manufacturing to design. In the reverse direction, a mechanism for the specification of the product from design into manufacturing was also required. The CalTech Intermediate Form (or CIF) [9] became a common representation that suited this purpose by specifying a layout, and thus indirectly specifying the masks that are used during IC fabrication.

Design Capability

The ability to effectively design VLSI circuits also required advances in representations, tools, and methodologies to aid in the design of increasingly complex circuits. While CIF provided a basic representation to get started with, innumerable other representations focused on different levels of the design problem were necessary and continue to evolve. New levels of abstraction (and their corresponding representations) have been identified that make possible the application of information technology to assist or automate design tasks. The availability of abstractions and representations spawned the introduction of computer programs to generate and manipulate these representations. Among the first of these were basic layout programs. These were joined by tools performing a variety of functions (*i.e.*, capture, synthesis, simulation, analysis, and verification). Soon, such tools became absolutely essential to the management and solution of design problems. In addition to the tools, ways of performing

VLSI design such that workable solutions could be achieved had to evolve. Whole new ways of thinking about and conceptualizing the VLSI design process were developed. VLSI design methodologies made clear ways to use the tools and representations in the quest for new circuits solutions. In short, all three of *representations*, *tools*, and *methodologies* were necessary prerequisites to the VLSI design revolution.

The Lesson for Process Design

The lesson for the growth of process design capability is clear. The evolution of representations, tools, and methodologies to improve and enable process design is essential. A well-defined interface to fabrication, and ultimately, the availability of inexpensive, application specific processing services are necessary. A great many software components and systems are needed to support and enable application specific process design and fabrication: new CAD and CIM systems are needed. Application specific processing is a radical vision, and one whose ultimate viability remains to be demonstrated. The approaches described in this thesis, however, can be expected to also benefit the design of general purpose and custom processes. If it is possible to design (in a short period of time) application specific processes that can then be executed in a programmable factory (with working silicon on first pass), then many of the same approaches may also be applicable to custom or general purposes processes, and improve design time and success rate of those processes as well. Many evolutionary advances are necessary to enable a revolutionary shift into application specific processing; but once the revolution has occurred, it will be hard to imagine semiconductor manufacturing without such a capability.

2.5 Summary

An *application-specific process* is one in which a designer specifies processing in addition to masking. Designers include not only process and device technologists (who now

monopolize this activity), but also circuit designers. Most immediately, these might include analog circuit designers, or, in the microelectromechanical domain, researchers or developers of sensors, actuators, and other micromachined structures. Examples of application specific processing range from the abbreviation of well-defined processes (as in the omission of an unneeded optional level of interconnect), to specification of some parameterized characteristic of the structures on the wafer (*e.g.*, the oxide thickness in a set of parameterizable capacitors) or devices (*e.g.*, the threshold voltage of parameterized nmos/pmos transistors), all the way to the wholesale definition of a new process (*e.g.*, the definition of a new micromechanical device). The achievement of application specific processing requires advances in the support of both *fabrication* (in the form of advanced CIM systems) and *design* (in the form of advanced technology CAD systems and frameworks [10] with enhanced design representations, tools, and methodologies).

Chapter 3

Computer Aided Fabrication

The MIT Computer Aided Fabrication Environment (CAFE) provides computer software and hardware support for all aspects of the design, analysis, development, planning, fabrication, and support of semiconductor manufacturing. A key belief adopted in this thesis is that the integration of design and manufacturing is essential; it is therefore important that the basic architecture of CAFE be understood, as it provides the philosophical as well as system context of this thesis.

3.1 Integration of Design and Manufacturing

An important goal of a computer aided fabrication (CAF) system is to improve upon *disjoint* design and manufacture, as illustrated in Figure 3.1. in order to achieve *integrated* design and manufacture as pictured in Figure 3.2. In integrated design and manufacture, the very boundary between these two activities is minimized. Communication is not only possible via a shared representation, there is also communication and cooperation between the fundamental design and manufacturing activities them-

†The CAFE system has been designed and implemented under the direction of D. Troxel and M. McIlrath; this chapter extracts and summarizes from descriptions of the system reported in [11] and [6].

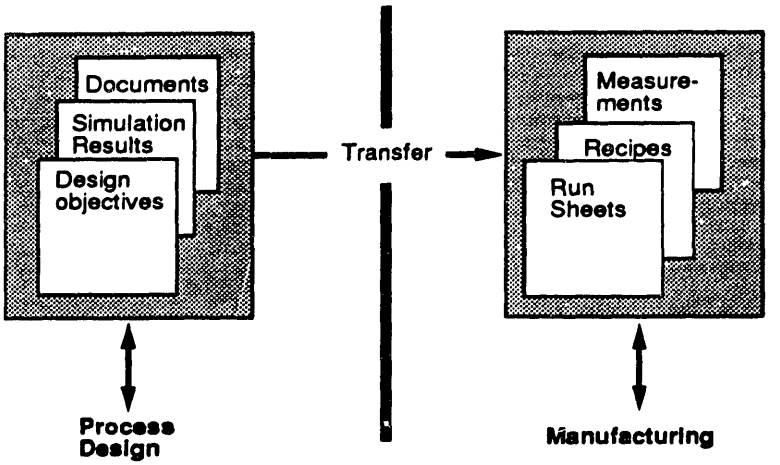


Figure 3.1: Disjoint Design and Manufacture.

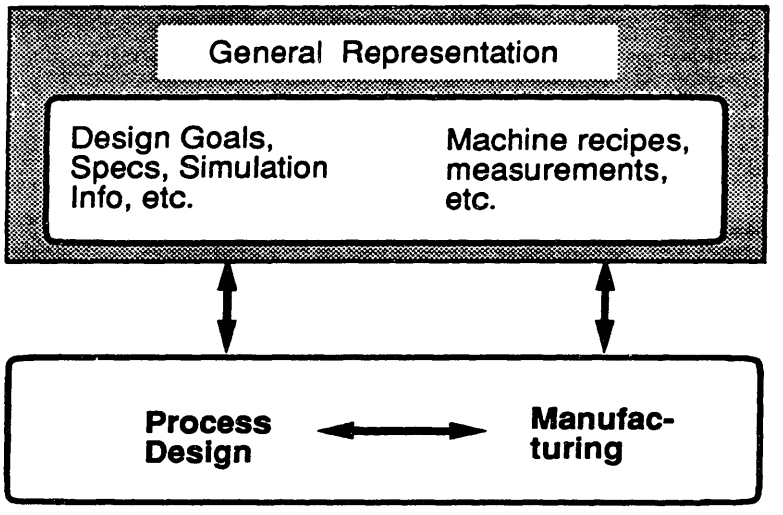


Figure 3.2: Integrated Design and Manufacture.

selves. Manufacturing makes use of both design data and design tools, and design makes use of both fabrication data and fabrication support tools. The manufacturing process is recognized to be constantly undergoing *design* via improvement and modification.

While such complete integration may seem radical, this is in microcosm how successful design and development already works. Semiconductor devices and fabrication processes are most often designed in a "development fab" where the design group is constantly making design decisions, investigating design possibilities via both modeling and experimentation, and improving the process or device. In many organizations, the design is declared "complete", and "transfer" to the manufacturing group finally occurs. Despite this declaration, however, the manufacturing group continues to "design" (though it is usually not recognized as such) the process and product during the normal course of process refinement, equipment replacement, and process control.

3.2 CAFE Architecture

The key step in the integration of semiconductor process design and manufacturing is the use of shared, common representations of the process and of the wafer (as pictured in Figure 3.3). In addition to this *data integration*, a hardware and software architecture is required to accomplish process design and manufacturing *activity integration*. The MIT Computer Aided Fabrication Environment (CAFE), depicted in Figure 3.4, has such integration as a key goal. CAFE serves two purposes: to support the design and manufacturing activities of the MIT Microsystems Technology Laboratories, and to serve as the hardware and software basis for computer integrated manufacturing (CIM) research.

The CAFE architecture consists of three parts. An *infrastructure architecture* defines and provides a set of domain-independent components for use within the system, including the Fabform user interface package [12] and the Gestalt object-

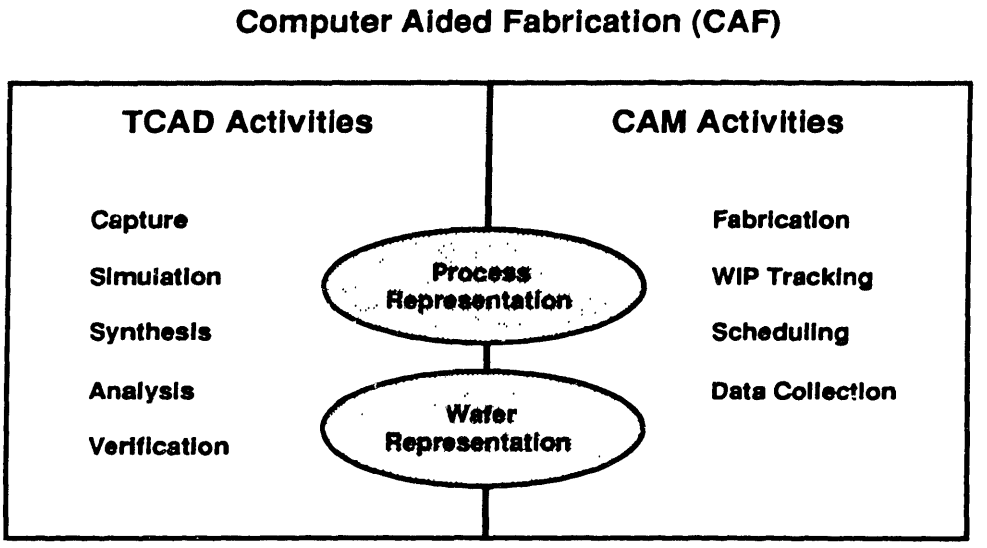


Figure 3.3: Integration of manufacturing and design activities via the use of shared representations of the wafer and of the process.

oriented database [13]. The tool and data *integration architecture* defines the conceptual schema and models used to represent and maintain knowledge and information about IC manufacturing within the system, and provides both user and programmatic interfaces to that information. Finally, CAFE *applications* provide support for a wide variety of activities, including not only process but also product, equipment, plant, and personnel management capabilities.

3.3 Process Applications in CAFE

The design, analysis, and execution of semiconductor processes is a key application area in CAFE. Data integration is achieved by a shared *process flow representation* (PFR). Activity or tool integration is achieved in part by a shared *process flow support* component which is used by both fabrication and development subsystems. The *fabrication support* subsystem is outside the scope of this thesis, and is only briefly summarized below. This thesis contributes to the *process development support* system

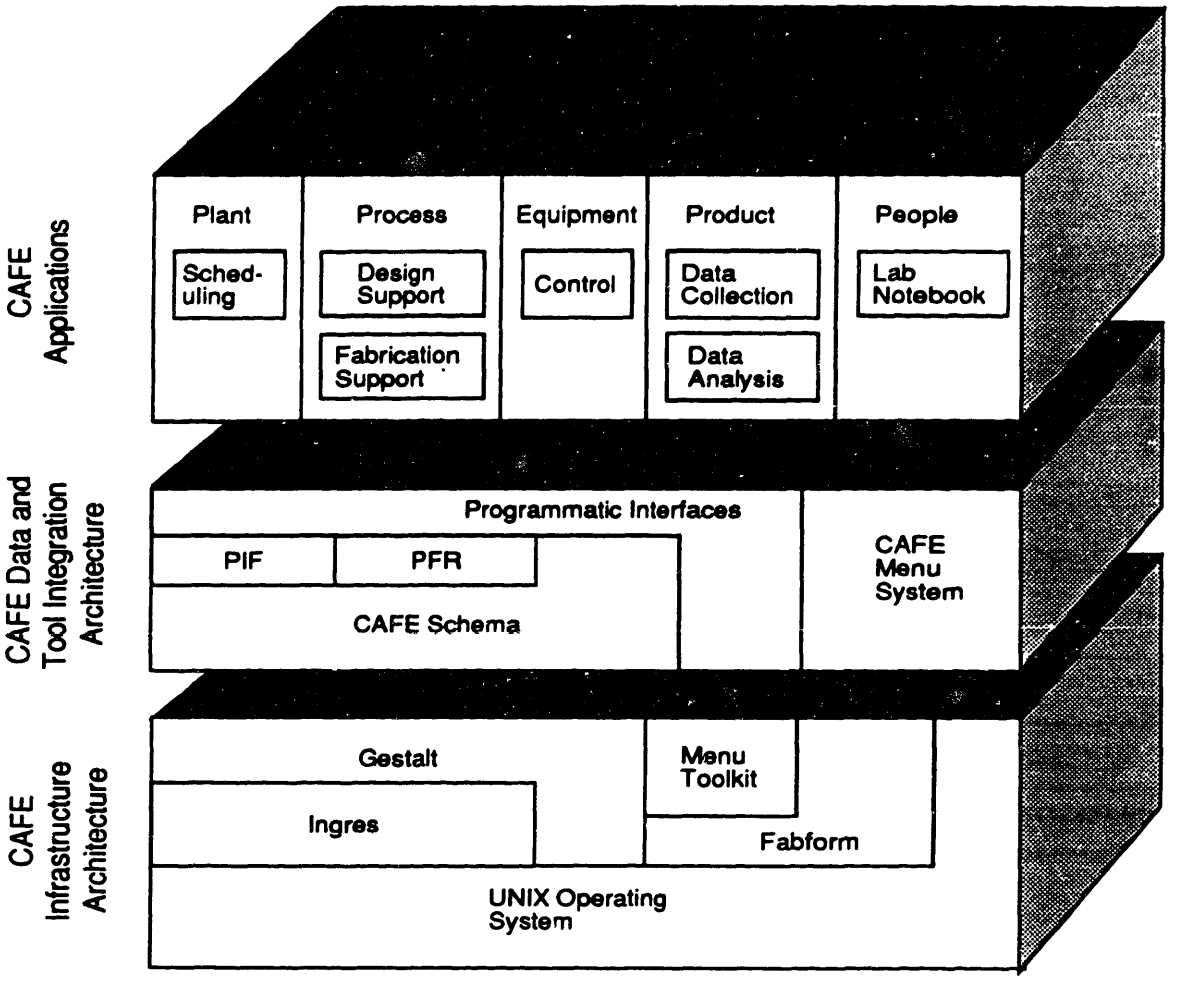


Figure 3.4: The MIT Computer Aided Fabrication Environment (CAFE).

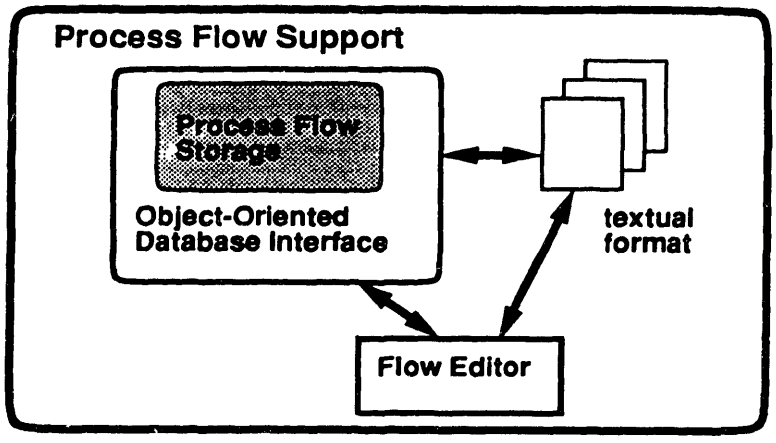


Figure 3.5: Implementation of the process flow support environment in CAFE.

in CAFE.

Process Support Subsystem

The process flow support subsystem consists of the elements shown in Figure 3.5. A PFR *reader* converts textual process flows into library and instance hierarchies of defined process operations and complete process flows in the Gestalt database [13]. The database provides shared, object-oriented access to PFR information for use by both design and manufacturing activities, and can be accessed and updated by all of the tools and applications in CAFE. A *process flow editor* provides a convenient user interface for generation, modification, and perusal of process flows [14].

Fabrication Support System

The manufacturing support subsystem [15] is shown in Figure 3.6. Wafer lots are attached to new or existing process flows. Reports such as work-in-progress (WIP) tracking are generated from process flow and lot information. Operations on single or multiple lots may be scheduled for eventual "execution" by the on-line fabrication system [16]. The PFR models of these processing operations are updated by the fabrication system, using results from measurements. Process development is thus

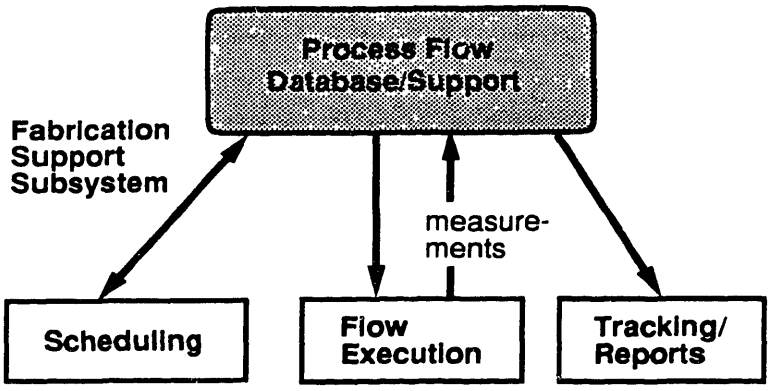


Figure 3.6: Semiconductor manufacturing support subsystem in CAFE.

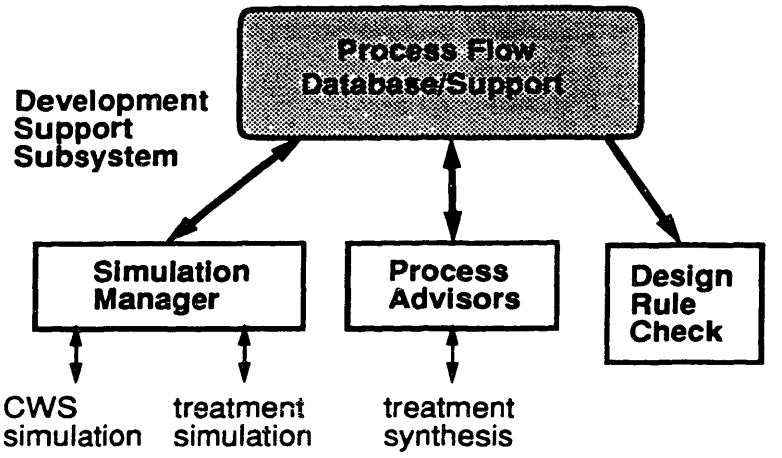


Figure 3.7: Semiconductor process development support subsystem in CAFE.

better informed by information generated during manufacturing. Design information and design tools are also available for use during process maintenance.

3.3.1 Process Development Support System

The CAFE process development support subsystem is illustrated in Figure 3.7. Process design tools, which themselves draw on the PFR support environment, provide help during process design, which takes place by incrementally generating a PFR for the process. Design may proceed on various levels, including wafer-state-change,

physical treatment, or machine settings. A *simulation manager* supports incremental simulation of the process during development. A *process advisor* aids in treatment level synthesis, providing analytic model-based estimates for the initial choice and modification of process parameters to achieve process goals. Other kinds of synthesis can be integrated into this architecture, such as recipe generation based on treatment and change-wafer-state information [17], [18]. By coupling synthesis tools with simulators (or experimentation) in an optimization loop, process parameters can be determined to necessary accuracy. A *flow library* of basic processing operations for available equipment is maintained in the data base with the necessary physical, change-wafer-state, and machine settings information. These operations may be accessed either by defined name or by specification; *i.e.*, by matching one of the attributes of an operation. The PFR can thus be used to support flexible process design styles that are capable of a variety of possible manufacturing implementations; for example, on different equipment types in the facility, or for portability to other fab lines. *Design rule checks* are available to test the adherence of processes to fabrication facility guidelines and policies before execution by the manufacturing subsystem.

3.4 Summary

The support of process development within the context of an integrated system for semiconductor manufacturing is the central focus of this thesis. The MIT CAFE system provides important infrastructural and integration support for the representations, tools, and methodologies that are described further in this thesis.

Chapter 4

Generic Process Model

This chapter presents a general framework for the modeling of semiconductor processing. On the most basic level, the intent of integrated circuit fabrication is to produce a wafer with specific electrical and mechanical characteristics, usually in the form of electronic circuits or chips, via some number of processing steps. The framework presented here consists of two parts. First, a *conceptual model* of semiconductor processing helps to categorize and structure the objects and interactions involved in a process step. Second, a *modeling methodology* emphasizes the description of both *state* information (such as the wafer state) and *transformations* (such as the changes to the wafer caused by fabrication equipment).

Models of semiconductor processing similar to those presented here underlie the development of a number of semiconductor process flow description and specification systems. The first of these was the FABLE language [20, 21], which introduced multiple *abstraction levels* for process information. The understanding of what process information to model, and the development of mechanisms for representing that information have continued to evolve in a second generation of process flow descriptions, including the MIT PFR [11] and the Berkeley Process Flow Language (BPFL) [22],

†This chapter is directly based on a collaborative paper by Boning, McIlrath, Penfield, and Sachs to be separately published [19].

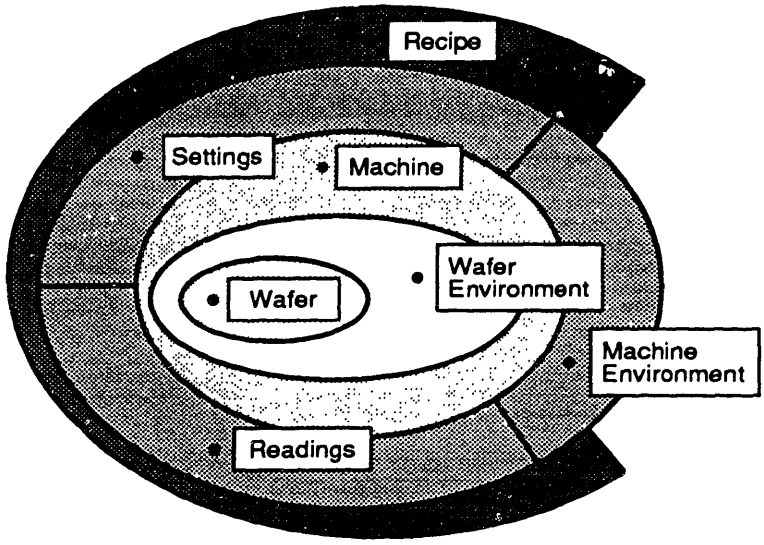


Figure 4.1: Conceptual semiconductor process step. A wafer is subjected to a physical wafer environment, which is generated by a machine. The machine provides settings for operating the machine, and meters for monitoring the machine, environment, or wafer. A control program or recipe dictates when and how to change settings, perhaps in response to readings. The machine resides within a machine environment or facility.

and within the Stanford MKS project [23, 7]. This chapter establishes a fundamental basis and terminology for the discussion, comparison, evolution, and development of semiconductor process models and process descriptions.

4.1 Conceptual Model

Semiconductor processing is conceptually pictured in Figure 4.1. During a process step, a wafer (or several wafers) is contained within some physical environment that has been generated by a piece of fabrication equipment as a result of machine settings, which are controlled or dictated by a program or recipe. The layering in Figure 4.1 indicates a number of interfaces: between the wafer and the wafer environment, the wafer environment and the machine, the machine and settings (as well as readings and the machine environment), and finally between settings/readings/machine en-

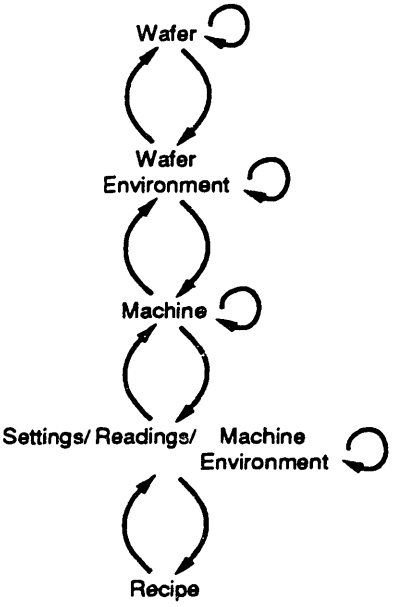


Figure 4.2: Conceptual chain of effects during semiconductor processing. An arrow from A to B is read as "A affects B".

vironment and control programs. This conceptual layering is loosely guided by the physical containment that exists during processing (*i.e.*, wafers within wafer environments within machines), but is more specifically motivated by a conceptual *chain of effects* among these levels as shown in Figure 4.2. One level can *affect* another only through an arrow shown in Figure 4.2 (corresponding to an interface of Figure 4.1), or through a chain of such arrows. Each level may evolve over time as it is affected by internal conditions, by the level above, or by the level below. The presence of three groups at essentially the same level of nesting in the diagram complicates the interactions and interfaces. The linear chain of effects in Figure 4.2 can be expanded into the more complicated diagram of effects in Figure 4.3 The intent of this modeling approach is to distinguish between these different levels, and to understand and describe the relationships (both fundamental and chained) among levels.

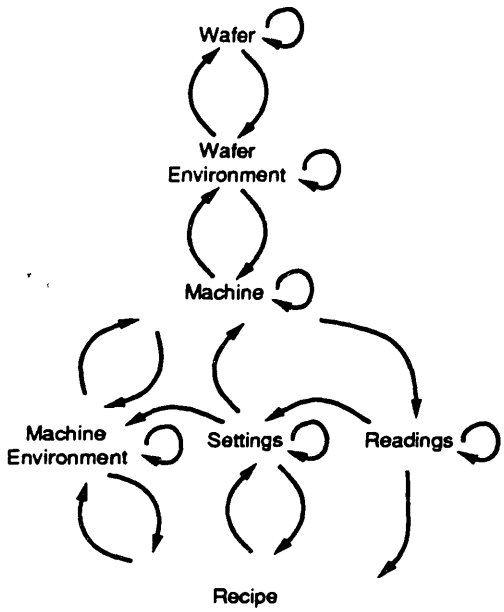


Figure 4.3: Conceptual chain of effects during semiconductor processing, with machine environment, settings, and readings separated.

4.2 Modeling Methodology

The terminology and diagramming methods used in this chapter are now introduced. A *state description* (or *state* set) is a set of state variables, and is denoted using an upper case letter (*e.g.*, Y). The knowledge of how a state set is affected by itself and other states is a *model*. A model can be described as a function or a *map* taking some number of input states to some number of output states, and is denoted using a lower case letter. For example, a model that takes X and Y state information as inputs, and produces X state information as output (pictured in Figure 4.4) can be denoted as a map $z : X \times Y \rightarrow X$. Functionally, one can say that an output state is the result of applying the map to the inputs, or $X_2 = z(X_1, Y_1)$, where $X_1, X_2 \in X$ and $Y_1 \in Y$, and the subscripts indicate sequentially occurring state descriptions. The model need not be *complete*; the model may only predict output states for a limited subset of elements on the input domains. The *directionality* of models (identification of input

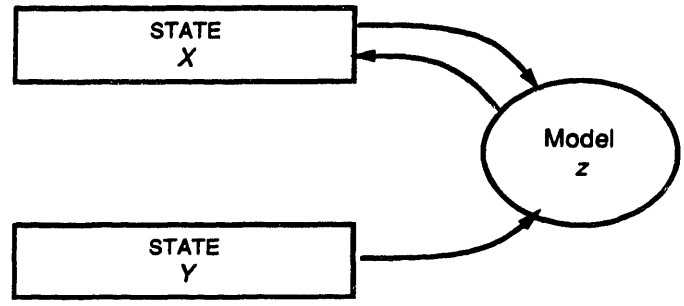


Figure 4.4: Example process state and model graph. Symbols are explained further in Figure 4.5.

and output states) is especially important in the modeling of physical processes, where a state A may affect a state B, but B may not affect A. As a result, a table (or some other set) of state pairs or tuples is not by itself a model; identification of the input and output states is also necessary in order to express the directionality of causal relationships.

A process step description may be depicted as a bipartite directed graph showing the dependencies, transformation, and flow of state information through models of physical (or other) mechanisms. As shown in Figure 4.5, state sets are depicted as rectangles, and models are shown as ovals. A state description may serve as the input or *source state* for zero or more models, and may be the output or *destination state* of zero or more models. A source model for some state description is defined as an *evolution model* for that state. The arcs in the state/model graph for a process step are interpreted as follows: any arc into a state A comes from a model m that can predict A given the availability of all the state information on arcs into m, provided that the model m is sufficiently complete (*i.e.*, that the mapping function is defined on the input state). Multiple arcs into a state indicate *alternative* (and not necessarily consistent) evolution models of that state. Multiple arcs into a model, on the other hand, indicate *required* state information. In the graph of Figure 4.5(a), either map $m_1 : B \rightarrow A$ or $m_2 : C \rightarrow A$ can be used to determine state A. Only in the graph of Figure 4.5(b) is there a single model $m_3 : B \times C \rightarrow A$ that combines the separate or

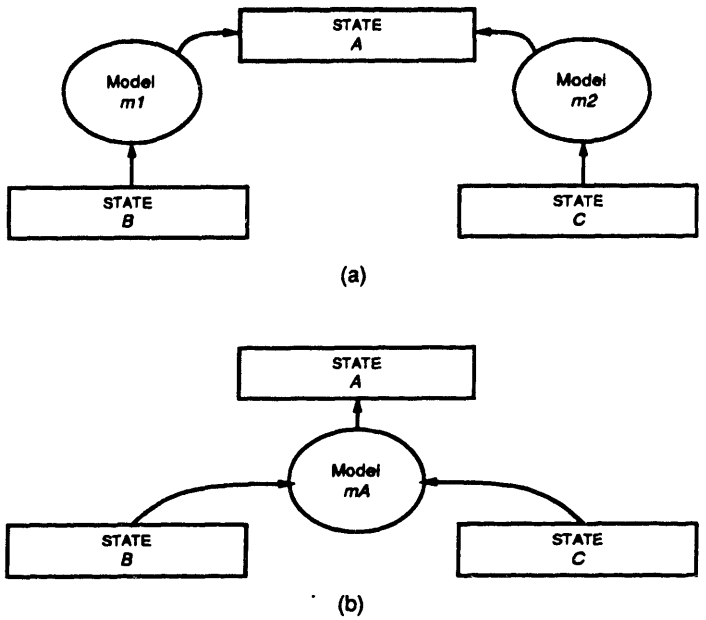


Figure 4.5: Process step state/model graph, where state sets are depicted as rectangular boxes, models are depicted as ovals, and arcs represent the possible flow of information. Two alternative models for predicting state A are present in case (a), while in case (b) a single model combines separate effects of B and C on state A.

cooperating effects of B and C on A.

State and model descriptions represent *categories* or sets of information. A state description may encompass any number of more refined and distinguished state sets which can interact with each other and with external states via refined models. A state description can be split as shown in Figure 4.6, so long as all necessary communication paths between states and models are retained. In addition, models may be split if there is no communication between the resulting models except by way of the connected state information (a model is driven only by external or internal state information, and not directly by other models). Models that produce only a single state description output are preferred but not required. Models that contain internal state may also be refined so as to make explicit the sources and destinations of its internal state information.

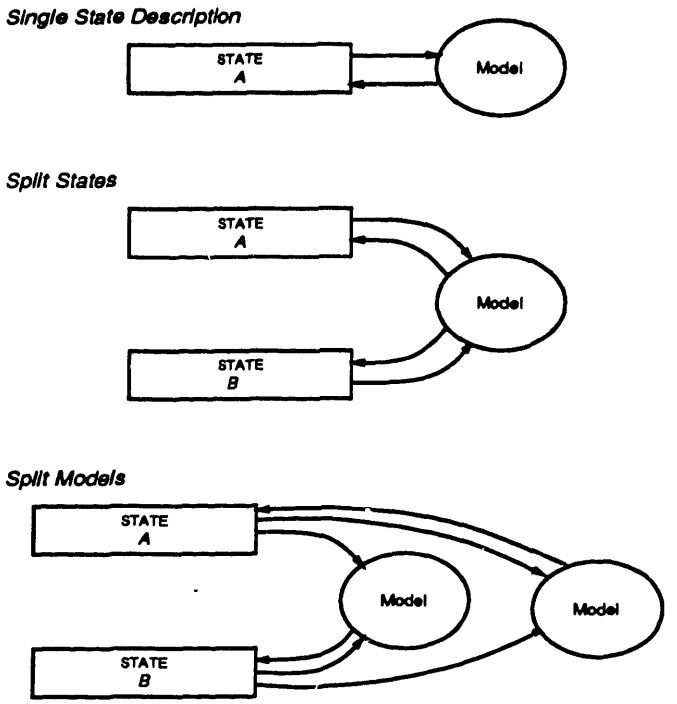


Figure 4.6: Splitting state and model information.

Several modeling concepts can be understood in terms of graph terminology. A node N_1 (either a state or a model) *depends on* or is *affected by* another node N_2 if there exists a directed path from N_2 to N_1. Two nodes are *connected* if there is a directed path between them. Two nodes are *directly connected* if (1) they are of different type and there is an arc between them, or (2) they are of the same type and there is a path between them passing through a single node of the opposite type. Nodes N_1 and N_2 are *coupled* if N_1 depends on N_2 *and* N_2 depends on N_1 (there exists a cycle containing N_1 and N_2). A process graph is *nonredundant* if no state in the graph has alternative evolution models. A *fully-connected* (where all allowed arcs are present), *nonredundant* process graph with n distinct state descriptions (each with at most one input) can have at most n models (each with at most n inputs). Even in such a nonredundant graph, the connection rules can lead to a large number of interconnections (up to $(n+1)*n$ arcs in the fully-connected nonredundant case).

A partitioning of semiconductor process state information that substantially reduces the number of connections between and *decouples* state and model information is described in the following sections.

4.3 Generic Process Models

A single comprehensive model of semiconductor fabrication would operate much like a finite state machine which takes as input all process state information, computes the interactions among these states, and produces as output a new state description. This very coarse view of semiconductor processing is refined further in the following sections to form *generic semiconductor process models*. This refinement proceeds using the methodology outlined above, including: (1) identification and splitting of process state information; (2) decoupling and splitting of process models; and (3) special simplifications and refinements for process modeling. Resulting generic process models describe the *structure* of process state and model information. Instantiation of the generic model via the introduction of specific state variables and transformation models gives rise to working process step models.

4.4 Process State Descriptions

The generic semiconductor process model identifies and differentiates among state information corresponding to the partitioning of Figures 4.1 and 4.2.

4.4.1 Wafer State

The state description of a wafer is potentially infinite in complexity and detail, and a wafer must be described in terms of some necessarily incomplete parameter set (denoted by $W(t)$, where t is time). Commonly used parameters for the description of starting material include crystal orientation, resistivity, and carrier type of a

wafer. Other state descriptions may include information about geometry (*e.g.*, layer thicknesses and other macroscopic and microscopic structure information), or scalar, vector, or tensor field information (*e.g.*, dopant concentrations, stresses, and other material and solid-state properties on the surface or within the wafer).

4.4.2 Wafer Environment State (Treatment)

The *wafer environment* is the physical environment, denoted $E(t)$, around the wafer. The wafer environment can be described in terms of waveforms (values as a function of time) of such parameters as temperature, partial pressures of ambient gases, the presence of liquids or chemicals near the surface of the wafer, or the fluxes of impurities or metallic compounds directed at the wafer. These parameters are thermodynamically intensive (defined for infinitesimally small regions of space, rather than depend on monolithic dimensions), a feature which helps to distinguish them from machine state parameters. The term *treatment* is often used to describe the environment for some period of processing time as a whole.

4.4.3 Machine State

The mechanical, electrical, or chemical composition and condition of processing equipment make up the *machine state*, denoted $M(t)$. This might include a description of the parts of the machine for purposes of equipment design, the current machine setup (*e.g.*, the gas plumbing for a furnace or the impurity source for an implanter), machine conditions during execution of a process (*e.g.*, the valve openings in a furnace, or the voltages across plates in plasma equipment), or a description of the degradation of the machine from run to run (*e.g.*, the buildup of silicon on a susceptor within an epitaxy reactor).

4.4.4 Machine Settings

The machine state and the *machine settings* provided by a piece of equipment are further distinguished. The settings, denoted $S(t)$, correspond to the positions of knobs or other controls of the machine. In general, settings may vary either continuously or discretely as a function of time in response to operator or automated instructions.

4.4.5 Machine Readings

Machine *readings*, denoted $R(t)$, are direct measures of machine state (and indirect measures of environment or wafer state). Examples of a reading are the current shown on a meter of an ion implanter, and the temperature from a furnace thermocouple.

4.4.6 Machine Environment

The machine also resides within a *machine environment* or fabrication *facility* from which materials (*e.g.*, gases, liquids, tool sets) flow. The facility may affect the machine through scheduling or disturbances (*e.g.*, humidity), and the machine may affect the machine environment through output wastes, equipment failures, *etc.*

4.4.7 Programs or Recipes

A *program* or *recipe*, denoted P, controls how machine settings are set or changed during a process step. Examples include recipe numbers which index tables of setpoints in furnaces, or written instructions to operators. A recipe might also be a computer program executed directly by the machine or a machine controller. A recipe is usually considered constant during any one process step (though explicit consideration of program state might be useful for simulating the operation of some control algorithm). A recipe might change, however, between process step executions (as in a *Run by Run Controller* [17]). Recipes or programs may also have limited control over the machine environment.

4.5 Process Model Descriptions

A causal effect shown in Figure 4.2 must be accomplished via some mechanism. In many cases, the mechanism is the laws of physics (such as fluid dynamics, electrodynamics, mechanics, or chemistry); in others, computational or human mechanisms are required. In this thesis, causal effects are described by by state transformation models as defined in Section 4.2.

4.5.1 The Generic Model

The central motivation behind the partition of process state described in the previous section is that it allows one to split process models into smaller (and potentially decoupled) models. The physical chain of effects introduced in Figure 4.2 reduces the interconnectedness of the state information involved in semiconductor processing. Each state category can only be affected by its own internal forces and by neighboring state categories as first shown in Figure 4.1. The resulting *generic semiconductor process model* is pictured in Figure 4.7. The necessity of physical causality suggests that the following is true:

1. The wafer evolves only under internal influences and the external action of its surrounding environment.
2. The environment around the wafer evolves under internal influences and the external action of a machine, and may potentially be affected by the wafer.
3. The machine state evolves through the internal workings of the machine, by the external influence of machine settings, and by interaction with the wafer and machine environments.
4. Machine readings are determined by previous readings and by the action of a machine.
5. Machine settings are determined by control programs, machine readings, and previous settings.
6. The machine environment is changed by the machine, the control program, and by external events in the facility.

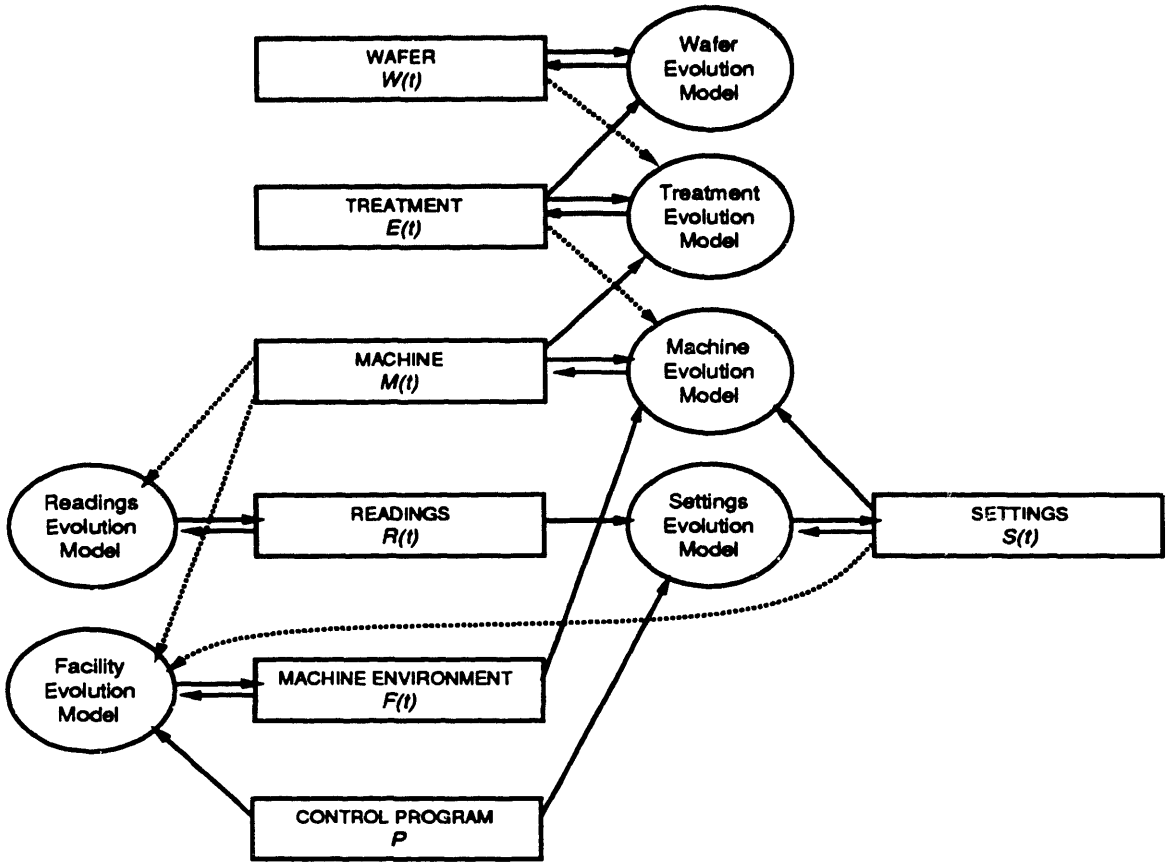


Figure 4.7: Modeling the evolution of process state in the generic semiconductor process model.

A requirement that physical action occur across ordered interfaces thus allows one to drop many of the potential interconnections between the states described in Section 4.4.

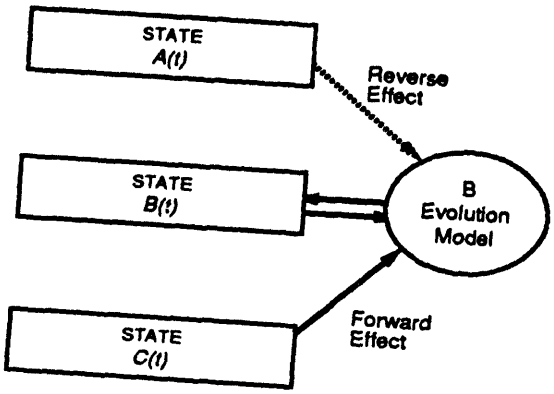
4.5.2 State Evolution Models

Each state discussed above may evolve with time. The forces driving the evolution of a state may come from two types of sources. First, a state may evolve under internal influences (its initial state). For example, a gradient in the concentration of gases in an environment will result in gaseous diffusion, which changes the position-dependent

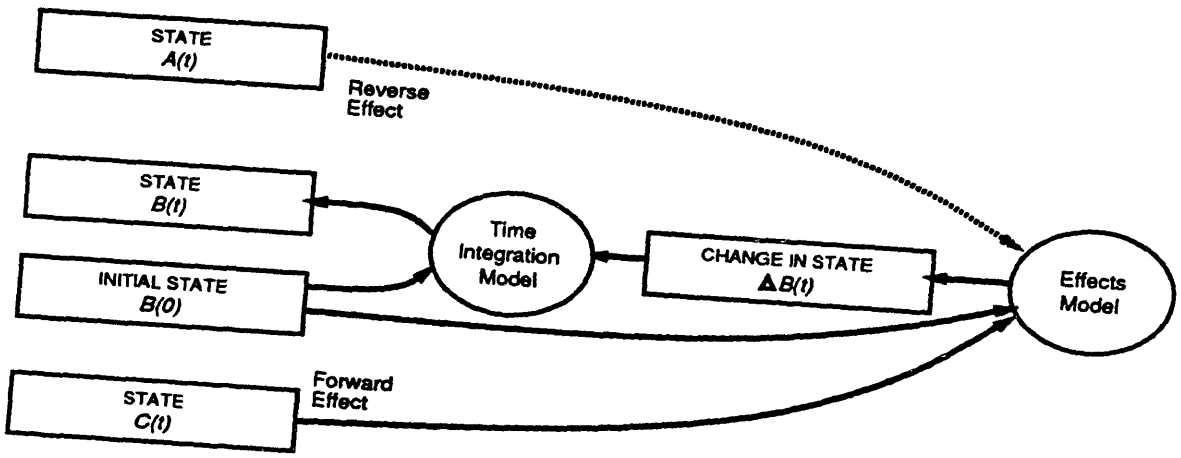
concentrations over time. Second, a state may evolve under the influence of external driving forces across the interface between the layers shown in Figure 4.1. In general, the effects across layer interfaces can be bidirectional (*e.g.*, the environment affects the wafer, and the wafer affects the environment). *Forward effects* are those working from the outer states in Figure 4.1 to the inner (or the solid arrows in Figure 4.7), and *reverse effects* (such as that of the wafer on the environment) are those directed in the opposite direction (and shown as dashed arrows in Figure 4.7).

The components of a state evolution model are shown in more detail in Figure 4.8. The reverse and forward effects, along with the initial state and internal effects, combine to act as agents of change to the initial state. It may or may not be possible to separate out the contributions to change due to each of these influences. For example, the change in the environment in general occurs by the wafer and machine acting together, so that the effects of the wafer and the machine on the environment cannot always be individually identified or summed. In these cases, the effects model must couple these contributions to produce the net change in the state. An *integration model* is also needed that describes how to integrate the initial state and change parameters over time to produce the output state.

Some terminology helps to distinguish between different approximate evolution models. When all three sources of state change are present, one can only say that a source state *affects* (the evolution of) the destination state. When only a single forward or reverse influence is present, one can say that the change in the destination state is the *effect* of the source state on the destination state. When a destination state has only a single source state, the relationship is even stronger, and one can say that the source state *generates* or *causes* the destination state. For example, if the wafer does not affect the environment, and the environment initial state is neglected, then the machine can be said to generate the environment.



(a)



(b)

Figure 4.8: The structure of the state evolution model in (a) is shown in more detail in (b). The initial state, forward effects, and reverse effects together produce a total *change in state*, which then operates on the initial state to produce the time evolving state.

4.5.3 Model Simplifications

For any process step, the generic model of Figure 4.7 can be instantiated and further manipulated through both refinement and simplification. For any process step, (1) additional state or model splittings may be appropriate, (2) many of the arcs connecting states and models can be ignored, and (3) mergings of models and states into simplified models may be appropriate. Expansion of a state evolution model as discussed in Section 4.5.2 is an important example of model and state splitting. The following assumptions and approximations may be used to eliminate arcs in a process step graph. First, one or more reverse effects may be ignored. For example, it is often the case that the wafer will have little reverse effect on the environment and can be neglected. Similarly, one or more forward effects may be ignored. In the case of measurements of the wafer by a piece of equipment, the effect of the machine on a mediating environment and on the wafer itself may be negligible. Third, the initial state is sometimes irrelevant. For example, a control program or recipe may specify all of the settings for a piece of equipment, so that the initial value of those settings is not important.

4.6 Basic Component Models

The generic process model of Figure 4.7 results in a number of decoupled component models. This section presents important further simplifications of several of these models. These basic component models are commonly used as building blocks in the construction of a process step graph.

4.6.1 Change in Wafer State

The more detailed evolution model of Section 4.5.2 can be used with any of the state sets in the generic process model. Because one often takes the output wafer from one process step and feeds it to the next process step in a complete process flow, the

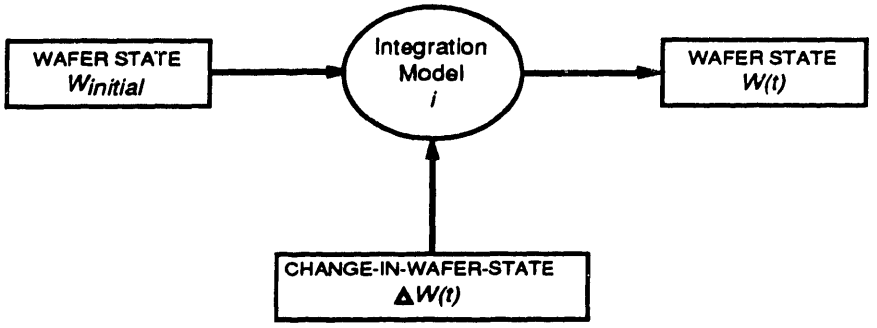


Figure 4.9: Change wafer state (ΔW) summarizes the effect a process step has on an initial wafer.

motivation for splitting out the initial and change in state is stronger in the case of wafer state than in other state descriptions. The state of the wafer for time $t > t_i$ can be described as a function of the state of the wafer at an initial time t_i and some number of *change in wafer state* parameters for time $t \geq t_i$. Thus it is useful to distinguish between *initial*, *change in*, and *resulting* state information, where the *change in wafer state* is the total effect a step has on the wafer. Just as the wafer state is a parameterized approximation of a real wafer, the change in wafer state is a parameterized description of the evolution of the wafer, and is denoted as *CWS* or $\Delta W(t)$. As pictured in Figure 4.9, the wafer state integration model is a map $i : W \times \Delta W \rightarrow W$. The *change in wafer state* may be time dependent ($\Delta W(t)$), or apply to the initial and final states only (ΔW). An example is the conformal deposition of an oxide layer, where the cumulative change in wafer state might be described as a geometric addition of material on top of initial surface topology. A corresponding time-dependent description of change in wafer state occurs when a material is deposited at a particular rate (*e.g.*, oxide is deposited on the surface of the wafer at one micron per hour).

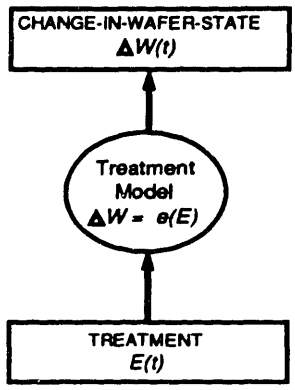


Figure 4.10: A treatment–CWS model describing the change in wafer state resulting from a given environment $E(t)$.

4.6.2 Treatment–CWS Model

A *treatment–CWS model* relates a treatment to a description of the change in wafer state resulting from the treatment (*i.e.* a map $e : E \rightarrow \Delta W$), as illustrated in Figure 4.10. The *treatment–CWS* model describes the *effect* of the environment on the wafer in the special case where (1) the effect is independent of the initial wafer state, and (2) the environment is not affected by the wafer. The essential input to the model is the environment $E(t)$, which may be known *a priori*, or may be calculated from a treatment evolution model.

4.6.3 Machine–Treatment Model

The term *equipment model* is often used to refer to any model describing the interactions of machine, settings, or control states and other states. In the generic process model, these different kinds of equipment models are differentiated.

A *machine–treatment model* relates machine state to environment state (*i.e.* a map $m : M \rightarrow E$). Often the interaction between the environment and the machine is tightly coupled; for example, the machine may have internal control mechanisms so as to maintain a specified environment. The machine is usually the most important

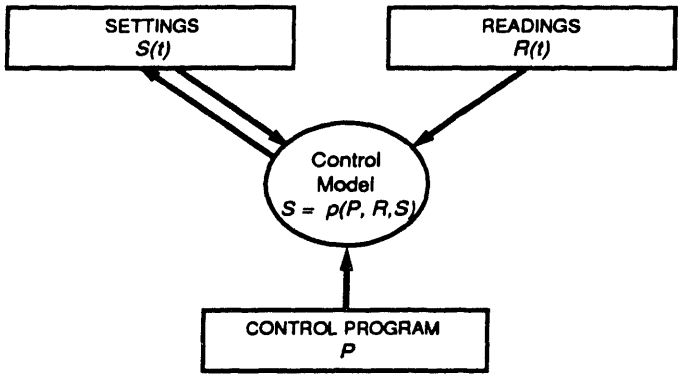


Figure 4.11: A control model describes the effect of a control program and readings on machine settings.

factor in the evolution of the environment. In the case where the reverse effects of the wafer on the environment are negligible, the treatment can be determined entirely by the machine model (the *machine–treatment model* directly predicts the treatment).

4.6.4 Settings–Machine Model

A *settings–machine model* relates the state of a machine to its external settings (*i.e.* a map $s : S \rightarrow M$). Such a model may be based on the mechanics, electronics, or other physics of the machine.

4.6.5 Control Model

A *control model*, illustrated in Figure 4.11, describes how settings change under the influence of machine readings as directed by a control program (*i.e.* a map $p : P \times R \times S \rightarrow S$). Machine readings may reflect in-situ machine, environment, or wafer state measurements (for real-time control), or pre- or post-process measurements (for feedforward or feedback run by run control). All measurements of machine, environment, and wafer state must physically occur through a reverse chain of effects (*i.e.*, one measures the wafer only via some impact of the wafer on the environment, which affects some machine state, which can then be read by some machine). The

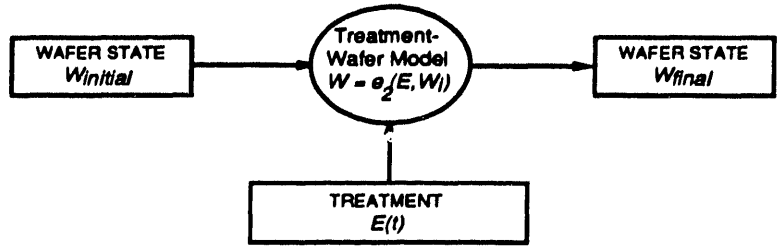


Figure 4.12: A composite treatment–wafer model directly describes the final wafer produced by an initial wafer and a treatment.

interpretation of machine readings to *measure* state requires either detailed fundamental models of these intentional reverse effects, or abstract models which map wafer, environment, or machine state directly to readings.

4.7 Abstract Component Models

The transformations described above are motivated by the actual chain of effects that occur during processing, and are basic incremental descriptions of how one state affects another in that chain of effect. It is often difficult or unnecessary to describe a process step in terms of these fundamental state and transformation components. For example, equipment that uses in-situ monitoring of the wafer state to control the process is often best described directly by a model that relates the settings to the change in wafer state. This section discusses such *composite* or *abstract* models.

4.7.1 Treatment–Wafer Model

The *treatment–wafer model* shown in Figure 4.12 describes the evolution of an initial wafer under the influence of a given treatment. Process simulators such as Suprem-III [24] and Suprem-IV [25] are programmatic embodiments of such treatment models. The *treatment–wafer model* does not separately or explicitly consider the step changes to the wafer; instead it maps an input wafer state directly to an output wafer state under the influence of the treatment without considering any "intermediate" change in

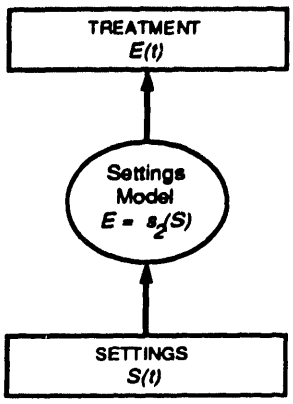


Figure 4.13: A composite settings–treatment model directly describes the treatment that results from machine settings.

wafer state. While internally most process simulators in fact do contain various kinds of change in wafer state descriptions (that is, treatment models may be internally *implemented* via more complex wafer state evolution models), one can conceptualize treatment–wafer models as in Figure 4.12. The treatment–wafer model is a map $e_2 : W \times E \rightarrow W$, while the treatment–CWS model is a map $e : E \rightarrow \Delta W$. Thus the application of e_2 can be considered a composite application: $e_2(W_i, E) = i(e(E), W_i)$ where i integrates the change in wafer state.

4.7.2 Settings–Treatment Model

A *settings–treatment* model, as illustrated in Figure 4.13, relates machine control parameters directly to the resulting environment (*i.e.* a map $s_2 : S \rightarrow E$). This is a highly useful model when the interior workings of the machine itself are not important, or when internal machine control loops work to accomplish a specified environment. The settings–treatment model becomes something of a "black box" by describing only the inputs (settings) and outputs (treatments) of the fabrication equipment. Calibration charts are examples of such settings–treatment models.

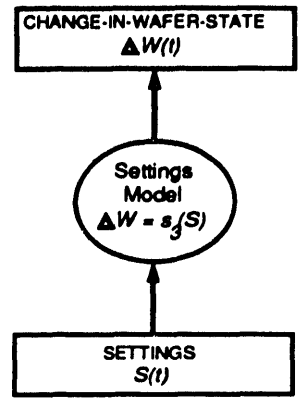


Figure 4.14: A composite settings–CWS model relating machine settings to the change in wafer state.

4.7.3 Settings–CWS Model

In addition to the fundamental *settings–machine model* and composite *settings–treatment* model, other composite settings models are possible. The first of these, illustrated in Figure 4.14, is a *settings–CWS* model. This model in essence considers the workings of *both* the machine and the treatment as a black box, and directly describes the change in wafer state resulting from settings (*i.e.* a map $s_3 : S \rightarrow \Delta W$). A final composite model is a *settings–wafer model* which directly describes the evolution in a wafer state resulting from the settings (*i.e.* a map $s_4 : S \times W \rightarrow W$).

4.8 Two-Stage Model

The *two-stage* process step model is a special case of the generic process model consisting of a subset of the states and models suggested in the conceptual model of Section 4.1. The two-stage model describes the effect on a wafer by decoupling and then *chaining* together the treatment and settings models described earlier, as shown in Figure 4.15. The critical condition for decoupling and chaining in the two-stage model is the existence of clearly defined interfaces that partition the wafer, treatment, and settings state variables into independent groups. The settings and environment

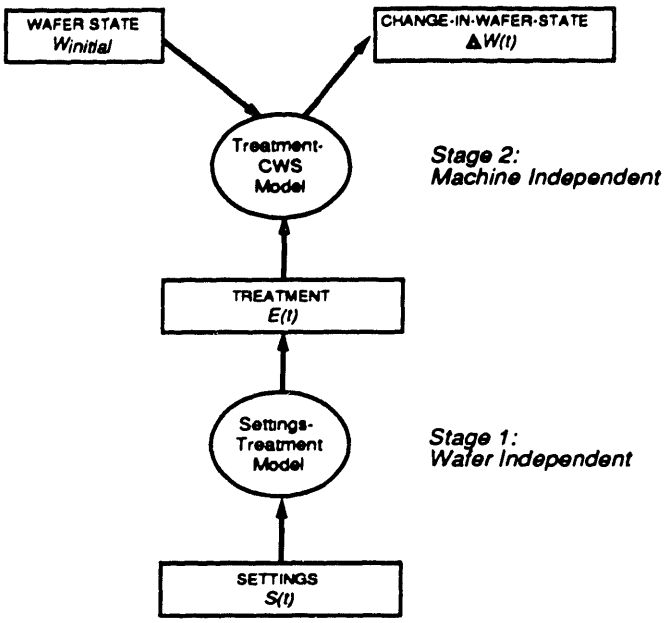


Figure 4.15: Two-stage process step model.

are readily distinguished. The boundary between treatment and wafer is sometimes located at the surface of the wafer, where the treatment might describe the material fluxes brought to bear on the wafer, and the treatment model then describes the effect of these on the wafer. In other cases, the boundary may extend some small distance from the wafer, and surface layer effects are considered within the treatment model. In general, the physical coupling between the wafer and the physical environment is bidirectional. When the *reverse* effect of the wafer on the environment is small, it is possible to model the environment solely as a product of the initial environment state and the machine (independent of the wafer), and to model the wafer state as a product of some machine-independent treatment and an initial wafer (independent of the machine or settings that produces the treatment). In this case, the chain of effects becomes unidirectional (from settings to treatment to wafer), resulting in the two-stage process model.

If the above conditions hold and a two-stage model of the step can be described, then a great deal of design and implementation flexibility is enabled. First, it becomes

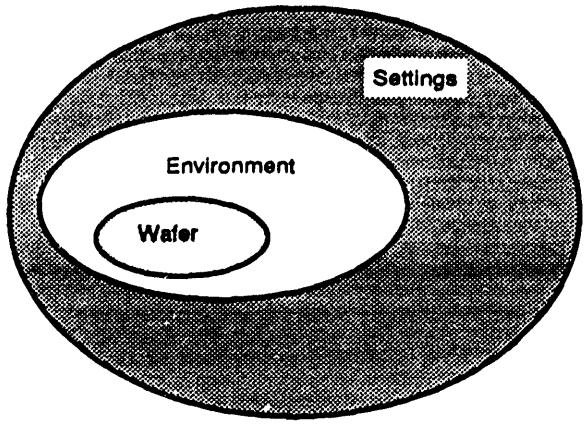


Figure 4.16: Conceptual semiconductor process step with machine state omitted. A wafer is contained and subjected to some physical environment, which is generated by machine settings.

possible to characterize, control, or specify a piece of equipment in terms of the treatment it produces independent of the specifics of the wafers that will be processed in the machine. Similarly, it becomes possible to characterize, explore, or specify process steps without concern for the specifics of the equipment. For example, one may want to consider the effect on a wafer of a temperature treatment before the particular piece of equipment is available or has been characterized. In the same way, the portability of a process step is enhanced: different machines might be used to implement a specified treatment. For example, the two-stage model is well suited for the machine-independent design of high temperature steps.

The two-stage model (and other process descriptions using abstract or composite transformations) can also be thought of as resulting from a less finely-grained partitioning of process state. The conceptual state partition corresponding to the two-stage model, for example, is pictured in Figure 4.16. That is, the complex workings of the machine state and interfaces to the machine state are hidden or *abstracted* by the settings to environment interface.

4.9 Additional Model Considerations

4.9.1 Temporal Composition

In principle, it is possible to describe the wafer state for all time, or to model the wafer state as it evolves over all time. During fabrication, there generally exist clearly identifiable process *steps* during which the wafer state is intentionally changed (*e.g.*, material depositions and etches, diffusions), and in between which the wafer should not change (as when the wafer is sitting in a buffer between workstations). It is often possible, therefore, to describe the state and model information for all time as a sequence of states and transformations that apply for specific time intervals. It is useful to do so in order that individual wafer changes, treatments, and settings can be more closely identified with each other. For each step in a sequence this composition in time can be repeated, leading to an arbitrarily long sequential description of state and model information.

4.9.2 State Description Conversions

It may be necessary to convert one state description into another description of the same state because different representations are more appropriate for computation, communication, or other uses. For example, a terse representation of the wafer state describing orientation, resistivity, and carrier-type must often be converted into a discretized (and more detailed) representation for use in process simulation. Conversely, it may be desirable to *characterize* a detailed wafer description resulting from simulation or measurements in terms of a more compact set of parameters. Multiple representations of other states in the generic process model also occur. For example, different process simulators now have *different* input languages that use different descriptions of treatments. Agreement on a single *standard* representation of process information by all consumers of that information is desirable but probably not completely realizable, so that conversion of state and state transformation descriptions

will continue to be necessary.

4.9.3 Statistical and Empirical Models

The generic process model is most easily understood in terms of *nominal* (*i.e.*, mean value) descriptions of the state information and deterministic models of how these states affect each other. As pointed out in [17], a spectrum of models is possible, from a mechanistic or physical basis on one extreme to entirely empirical on the other. In addition, models that include statistical descriptions and dependencies of states add an important dimension to the generic process model. In current practice, models of the fundamental transformations are often physically motivated and provide only nominal descriptions of the wafer, while abstract or composite transformations tend to be more empirically based (and sometimes provide statistical in addition to nominal information). Statistical models tend to be empirical as a result of the difficulty of adequately modeling statistical dependencies using physically-based models.

4.10 Oxidation Example

In this section, the use of the generic process model to categorize information about a thermal oxidation unit process step is illustrated. The state/model graph summarizing the oxidation step model is shown in Figure 4.17.

4.10.1 Wafer States

A one-dimensional description of the initial wafer state for the oxidation step is that W_i is a p-type silicon wafer with <100> orientation, and resistivity 20 Ω-cm. This is an incomplete description of the initial wafer state; it does not, for example, describe two- or three-dimensional characteristics of the wafer, such as its diameter or thickness, defect densities, impurity doping, etc. For this oxidation step, the resulting wafer state W_f can be described as a <100> p-type silicon wafer, with 474Å of silicon-

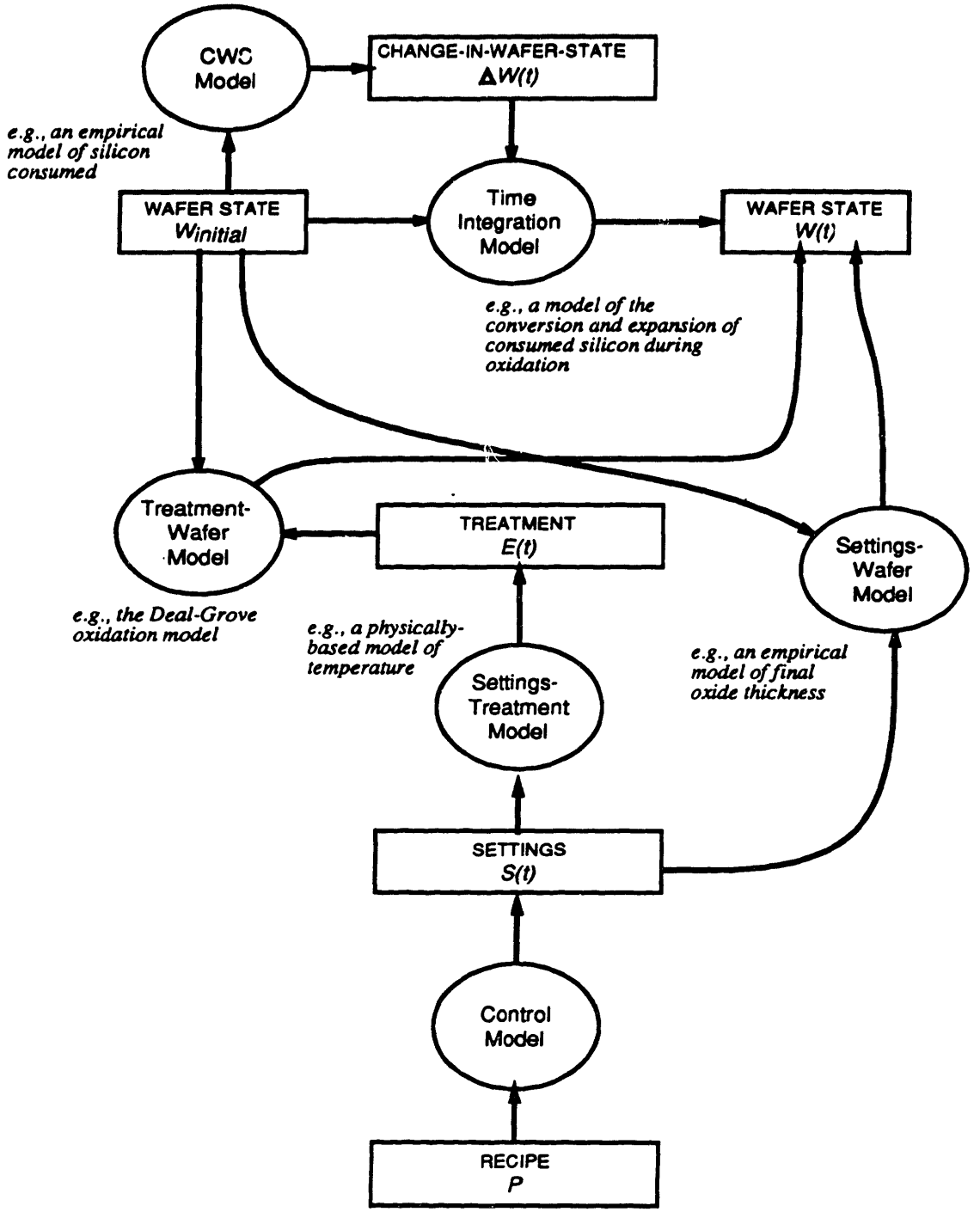


Figure 4.17: State/Model graph describing of a thermal oxidation process step.

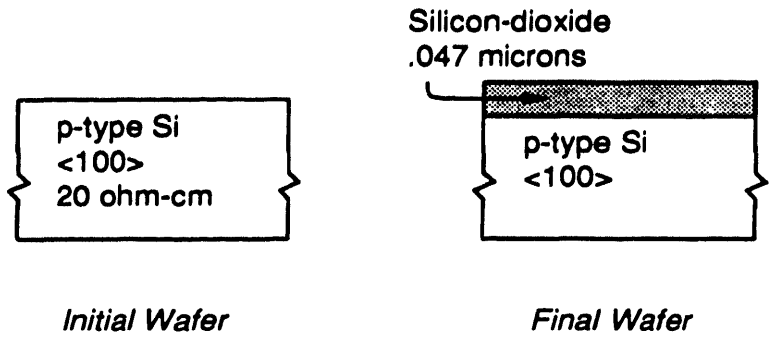


Figure 4.18: Pictorial descriptions of the initial and final wafer states for a thermal oxidation process step.

dioxide on the surface. The initial and final wafer states are shown schematically in Figure 4.18.

4.10.2 Environment State

A description of the environment state during the process step is the set of time-varying waveforms shown in Figure 4.19, where the temperature and ambient gas partial pressures at the surface of the wafer are depicted. Again, this description is by necessity an approximate (and incomplete) description of the real environment that the wafer sees. The value of such a description lies in its usefulness. For example, the description may be sufficient for use in a treatment model to calculate resulting wafer characteristics. The graphically depicted waveforms of Figure 4.19 may have corresponding textual or mathematical descriptions. Programs such as Suprem-III, for example, use a textual description of constant or ramped temperature and pressures for the piecewise-linear description of some treatment parameters.

4.10.3 Settings and Recipe

The settings for the oxidation is a detailed schedule of gas flow rates, wafer boat push and pull rates, temperature control ramps, etc. For this example, the settings are

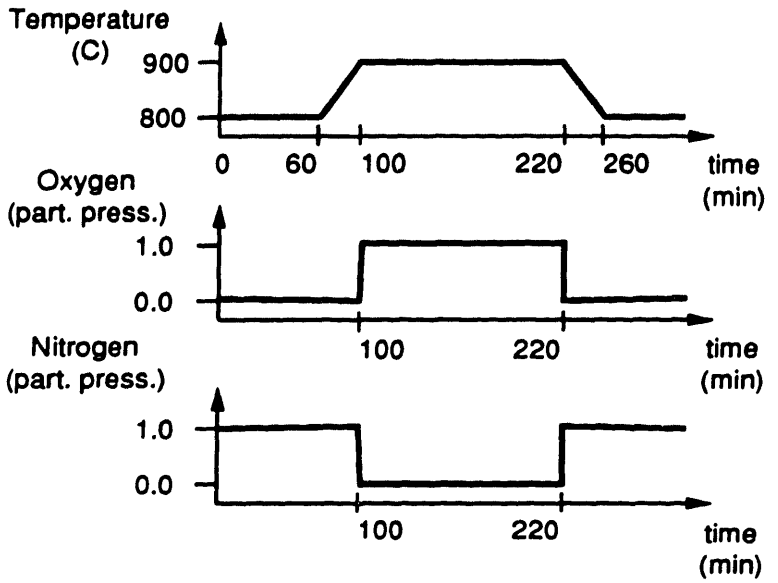


Figure 4.19: Description of the treatment during the oxidation step.

specified implicitly by a recipe number (the control program for the step), with the exception of the wafer push and pull rates which are separate, explicit settings.

4.10.4 Change in Wafer State

A description of the change in the state of the wafer is that 209Å of the silicon on the surface of a <100> oriented silicon wafer is converted to silicon dioxide with a material expansion factor of 2.27. A *geometric integration* model then uses these parameters to operate on an input wafer state description to calculate the final wafer state (resulting in 474Å of oxide). The change in wafer state, in combination with the integration model, describes a treatment-independent algorithm for the calculation of a new wafer state given an original wafer state.

The change is wafer state parameters can also be calculated via a change in wafer state model, given the initial wafer state. In this case, the CWS model is the table as shown in Figure 4.20, where the amount of silicon converted to oxide (with further material expansion) is assumed to be a function of the initial oxide thickness.

initial oxide (Å)	converted silicon (Å)
0	209
10	205
50	191
100	176
150	165
200	157
300	146
500	132
750	119
1000	109

Figure 4.20: An empirical change in wafer state model relating the amount of surface silicon converted to oxide during an oxidation. The final wafer state can then be calculated from this parameter.

4.10.5 Treatment–Wafer Model

A state triplet $(W_i, E(t), W_f)$, where the wafer and environment states are defined as above and W_f is the output state, can be thought of as a single-point sample of a more general treatment–wafer model, which could in principle consist of the set of all triplets (W_i, E, W_f). In some cases, however, the initial wafer state may not be known beforehand, and a treatment model that enables the calculation of the final wafer state given various different initial wafer states and treatments can be very useful. For the oxidation of silicon, the well-known Deal-Grove model [26] corresponds to such a treatment model. The model is often expressed as:

$$x_{ox} = \frac{A}{2}\left\{\left[1 + \frac{(t+\tau)4B}{A^2}\right]^{1/2} - 1\right\} \tag{4.1}$$

and

$$\tau = \frac{x_i^2 + Ax_i}{B} \tag{4.2}$$

where x_i is the initial oxide thickness, x_{ox} is the final oxide thickness, t is the oxidation time, A is the parabolic rate constant, and B/A is the linear rate constant. Both A and

B/A are expressed as a function of temperature and the ambient gas, as well as other parameters in more complicated versions of the model. The common characteristics of a treatment model appear here: (1) a dependence on the initial wafer state (here x_i), and (2) a dependence on the physical environment that the wafer sees over the duration of the process step (here temperature and ambient gas).

More complex treatment models are often contained within process simulation programs such as Suprem-III [24] or Sample [27, 28]. These models, in conjunction with program-specific descriptions of initial wafer state and treatments, and the computational mechanisms of the programs, combine to calculate simulator-specific descriptions of the evolving (or final) wafer state.

4.10.6 Control Model

In this oxidation example, the external control model is very simple. It is assumed that a human operator will be given instructions to start the process step, and will in turn instruct the furnace controller to use recipe 210 for the process step. The machine controller, then, looks up recipe 210 to find all of the temperature, pressure, gas flow, and other settings for the step. It is further assumed that these settings (or *set points*) will not be changed by the operator or the controller during the process setp.

4.10.7 Settings–Treatment Model

Once again, the pair $(S, E(t))$, where S is the input and $E(t)$ the output state descriptions described above, is a one point sample (or a single row table) of the settings–treatment model for this oxidation step: it states that the settings corresponding to recipe 210 generate the associated treatment. For our oxidation example, a second settings–treatment model provides a more accurate description of the temperature that the wafer sees during the push or pull of the wafer boat from the center of the furnace by considering the position dependent temperature as the wafer moves

Push-rate (m/min)	Pull-rate (m/min)	$\overline{x_{ox}}$ (Å)	σx_{ox} (Å)
2	2	474	2
2	3	473	3
3	2	473	3
3	3	474	4

Figure 4.21: An empirical equipment model relating mean and standard deviation of wafer oxide thicknesses to push and pull rates.

through the temperature zones of the furnace. One could use this model to calculate a more complete treatment description to replace the waveforms of Figure 4.19, and to achieve some parameterization and generalization of the model in terms of the input settings. It is important to note that explicit description of both the actual control program (internal to the furnace controller) and the machine state are elided.

4.10.8 Settings–Wafer Model

As in the other examples, the triplet (S, W_i, W_o) where W_o is the output state implicitly defines a simple equipment model. An extension of this *empirical* style of model building is the expression of many such triplets, or the reduction of many such samples into a statistical description of the relationship between output wafers, input wafer, and given settings. For example, one may wish to consider the uniformity of the oxide thickness across a given wafer. One may measure the oxide thickness at several locations across the wafer and evaluate the average and standard deviation of thickness. An interesting settings–wafer model might relate the statistical description of oxide thickness (perhaps accumulated over many executions of the process or many similar wafers) to the push and pull rates of the wafer, keeping the rest of the recipe constant. An example of such an empirical equipment model is shown in Figure 4.21.

4.11 Use of the Generic Model

The generic process model (the forward directed components of which are summarized in Figure 4.22) identifies several ways to describe the transformations a wafer experiences during processing: (1) directly via the change in wafer state; (2) indirectly by combination of a treatment and a treatment model; or (3) indirectly via a cascade of other state and model information. The generic model extends previous process representation work [20, 21, 22, 7, 29] in several ways. First, the distinction between state and state transformations has been introduced. Secondly, explicit consideration of wafer state is recognized to be necessary. The model distinguishes between and notes the need for both *fundamental* model relationships and *composite* relationships which bypass intermediate state and model descriptions. Finally, a methodology for the definition and manipulation of process state and model information has been introduced. Useful states and state transformations in the generic process model are summarized in Figure 4.23.

Since the *two-stage* model was first proposed by Penfield in 1984, the generic process model has formed the basis for a great deal of discussion and software development within the MIT Computer Aided Fabrication (CAF) project. The model is useful in clarifying and supporting the use of process information in numerous activities including process representation, simulation, synthesis, and control; these are summarized below.

4.11.1 Process Representation

Any one of the above states or models might be considered a partial *process specification* (or description of the process sufficient to support some activity). Several or all of these states and models, or indeed multiple variants of each, however, are often desirable to provide as much information about a process step as possible. Some subset of states and models may be sufficient for some activities but incomplete for

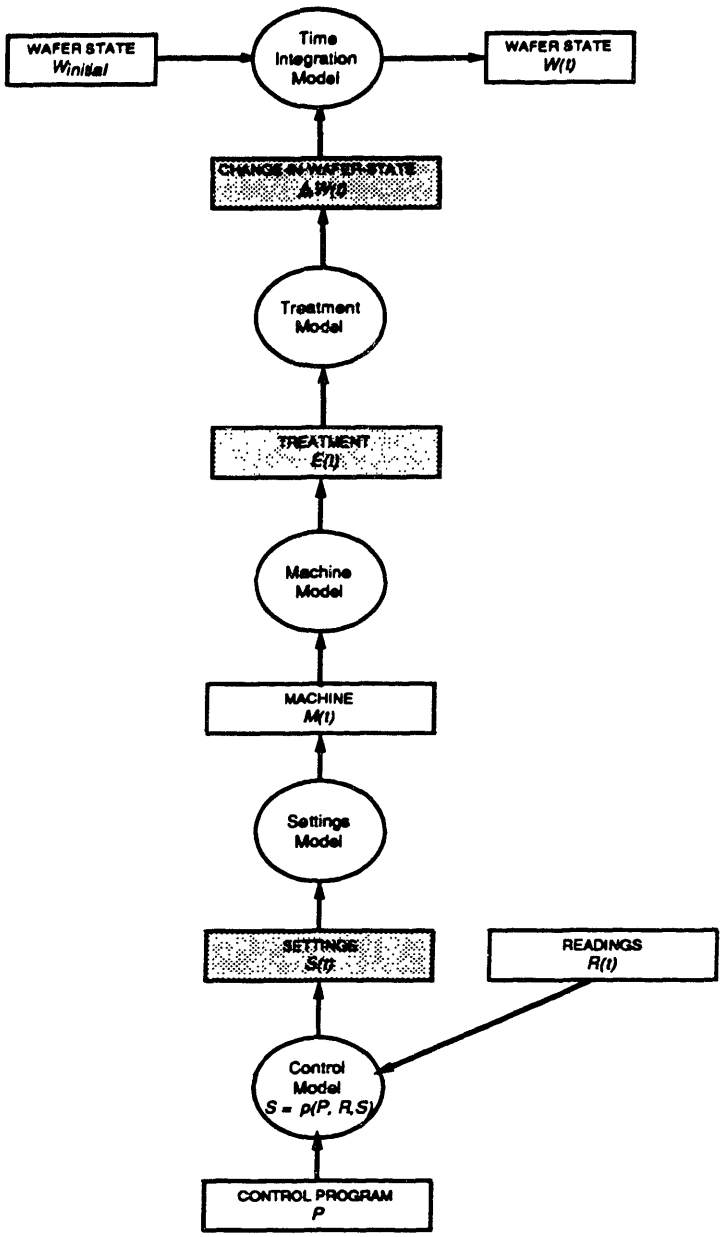


Figure 4.22: Forward directed generic semiconductor process model. Sets of state variables are shown in rectangular boxes, and state transformations are shown in ovals. The components that are lightly shaded can be directly described in the MIT Process Flow Representation [11].

State Sets	
W	Wafer
ΔW	Change in Wafer (CWS)
E	Treatment
M	Machine
S	Settings
R	Readings
P	Programs
Basic Component Models	
$i : W \times \Delta W \rightarrow W$	CWS integration
$e : E \rightarrow \Delta W$	treatment–CWS
$m : M \rightarrow E$	machine–treatment
$s : S \rightarrow M$	settings–machine
$r : M \rightarrow R$	readings
$p : P \times R \times S \rightarrow S$	control
Abstract Component Models	
$e_2 : E \times W \rightarrow W$	treatment–wafer
$s_2 : S \rightarrow E$	settings–treatment
$s_3 : S \rightarrow \Delta W$	settings–CWS
$s_4 : S \times W \rightarrow W$	settings–wafer
Two-Stage Model	
W, ΔW, E, S, and models e and s_2	

Figure 4.23: Typical component states and state transformations in the generic process model.

others. The components of the generic process model described above are not exhaustive: additional state and model information may be desirable or necessary. Because understanding and knowledge about a process step is always incomplete, process flow representations or languages based on the generic model (either directly or indirectly) should not require or mandate the presence of a *complete* process description. For example, the strict requirement that *effects* be *implemented by* treatments which are *implemented by settings* in FABLE [21] imposes restrictions and dependencies that made the language difficult to use. Process representations should, however, provide mechanisms for the expression and structuring of as much process information as possible, including *both* state and model descriptions.

The MIT Process Flow Representation (PFR) [11] is a unified, computer-manipulable description of process information which underlies much of the CAFE system [6]. The generic process model provides a theoretical basis for several operation attributes in the PFR, including the *change-wafer-state*, *treatment*, and *settings*. The generic process model also necessitates the description of wafer states. A uniform wafer representation using the *Profile Interchange Format* (PIF) has been developed(Chapter 6).

4.11.2 Process Simulation

Programs such as Suprem-III calculate the evolution of wafer state given two categories of process state information (in addition to initial wafer state). The real value of the program is in the calculation of the diffusion of impurities in silicon and other materials, as well as the growth of oxides at elevated temperatures. For these steps and others involving high temperature furnace processing or ion implantation, treatment information is required. For other steps, including etching and deposition, Suprem-III depends on simple descriptions of the desired change in wafer state, and uses this information directly to evolve the wafer structure.

The generic process model, along with uniform descriptions of its components in a process flow language, provides a framework for the specification of process infor-

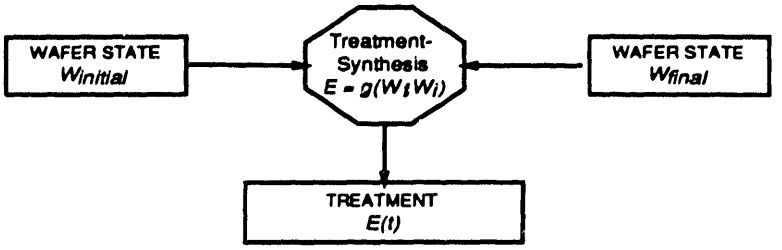


Figure 4.24: Process synthesis involves different transformations of the process state, indicated here as a hexagon.

mation in a simulator-independent fashion. The automatic generation of simulator input from such descriptions is an active area of research [6, 29, 30].

4.11.3 Process Synthesis

The generic process model describes physical transformations of process state. Other kinds of transformations of process state are also possible, particularly during process design. Process simulation might involve the calculation of $W_f = e_2(W_i, E(t))$, where W_f is the final and W_i the initial wafer state. *Process synthesis,* on the other hand, might involve the determination of the treatment necessary to produce a desired wafer state ($E = g(W_i, W_f)$) as shown in Figure 4.24. Such synthesis transformations may be constructed in part from the causal models of the process. In a few cases, the form of the model $e_2(E)$ is sufficiently simple that direct calculation of $g = e_2^{-1}$ is possible [31, 32]. For example, in the oxidation treatment model of Section 4.10.5, if the temperature and gas are fixed, one can calculate the oxidation time as:

$$t = \left[\left(\frac{2}{A}x_{ox} + 1\right)^2 - 1\right]\frac{A^2}{4B} - \tau \tag{4.3}$$

Given a large library of oxidation step descriptions including many (W_i, W_f) pairs, it may also be possible to search for an operation that already satisfies e_2^{-1}. In other cases, the synthesis transformation might be implemented via numerical optimization in order to determine the treatment [30], as illustrated in Figure 4.25.

In a similar fashion, one may wish to perform other synthesis functions, such as the

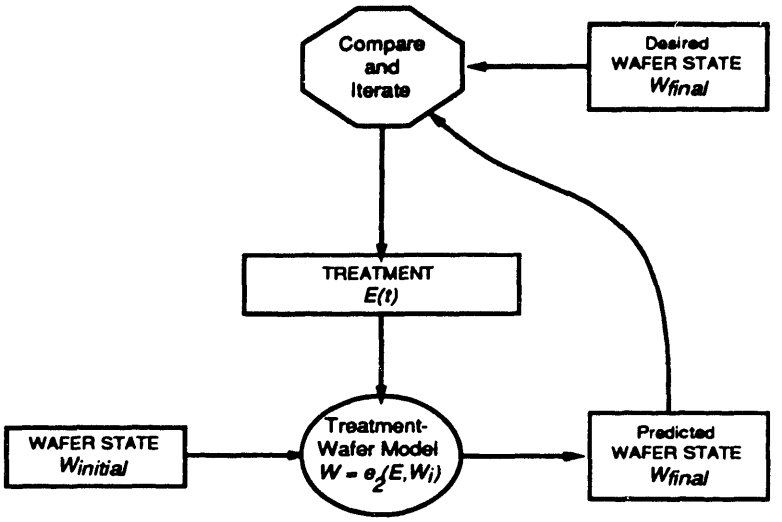


Figure 4.25: Process synthesis involves different transformations of the process state, indicated here as a hexagon.

calculation of $S = g_2(E(t))$. This corresponds to *recipe generation,* and may require numerical simulation, experimentation, and optimization of settings and equipment. The application of expert system technology to process synthesis and recipe generation has been demonstrated [33], where heuristic reasoning, formula solving, table lookup, and external simulation are all used to generate combinations of treatment, machine state, and settings required to achieve specified wafer state changes in the case of polysilicon deposition.

4.11.4 Process Control

The control of semiconductor processes is critical in semiconductor manufacturing. Process control is conventionally treated as shown in Figure 4.26, where disturbances as well as product and process parameters are inputs, and the output product is monitored and feedback via process parameters occurs [34]. The generic process model can be related to process control as shown in Figure 4.27. The specific inputs to the manufacturing process have been identified using the generic process model. State

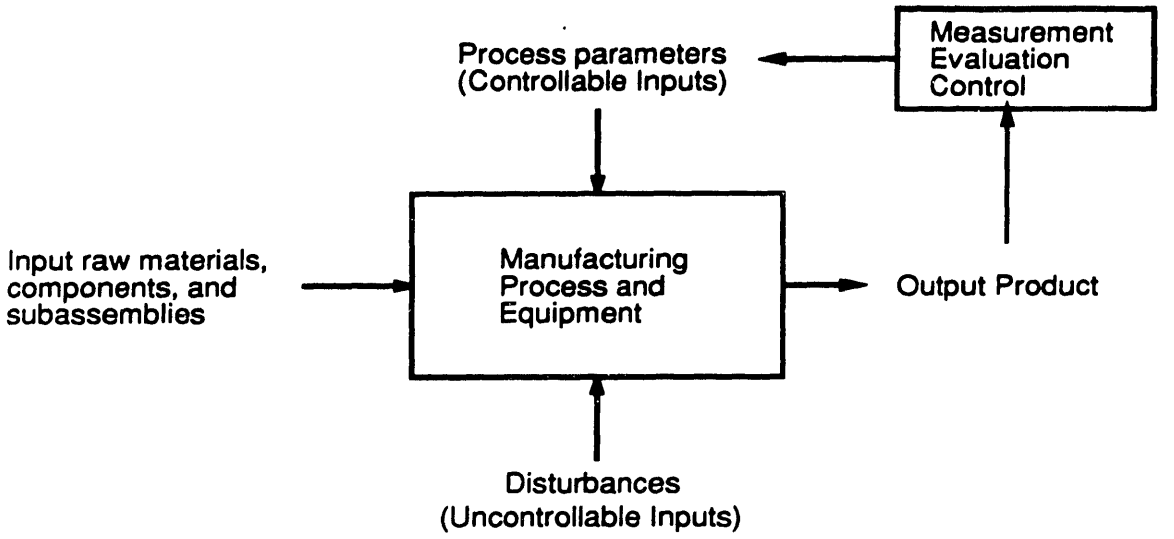


Figure 4.26: Conventional process control [34].

information, such as the wafer environment, has both controllable and uncontrollable components. The process may result in not only output product (wafer state), but may also result in changes to the machine, facility, or other states. The control of the process depends upon the development of programs and methodologies for monitoring the process via readings, and affecting the process state via settings.

An architecture suited to the needs of process improvement and yield enhancement, as well as process control, has been proposed [17]. Aspects of that architecture can be related to the generic process model and Figure 4.27. First, the process control architecture focuses on descriptions of not only wafer state but also variability in the wafer state from wafer to wafer and run to run. That is, process control depends fundamentally on a statistical model of processes. Second, a methodology for the explicit creation and maintenance of statistical models of process state transformations is proposed. While modules such as a *Run-by-Run Controller* focus on direct equipment models, other modules within the architecture (such as the *Flexible Recipe Generator*) consider degrees of physically based equipment and treatment models. Finally, control algorithms that change not only settings but also recipes between

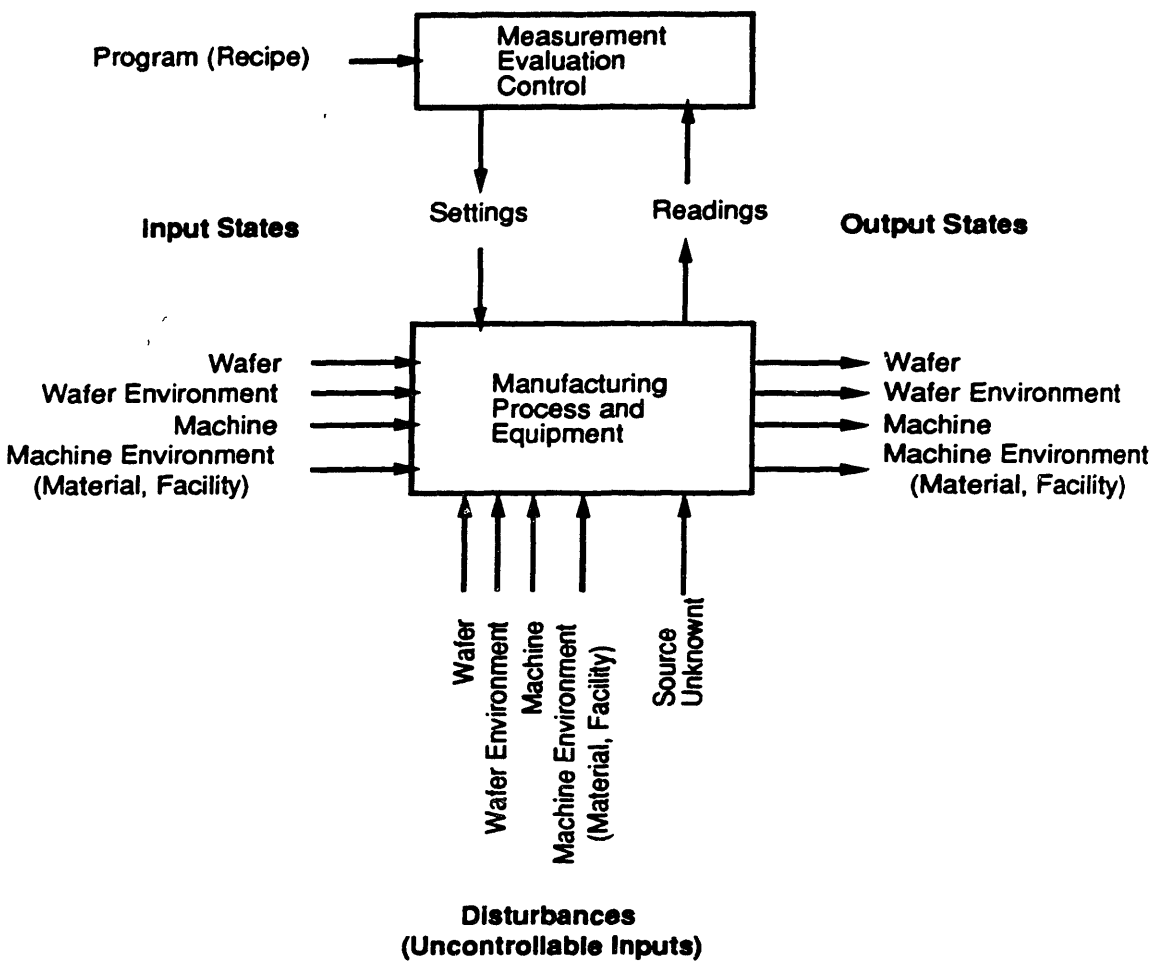


Figure 4.27: Process control under the generic process model.

runs of the process have been demonstrated.

4.12 Summary

A methodology has been introduced for modeling the state and transformations of state that occur during manufacturing of integrated circuits. The methodology has been applied to describe categories of information that are important in semiconductor fabrication, although the methodology may also be applicable in other domains of manufacturing. The resulting *generic semiconductor process model* aids in the conceptual understanding of fabrication processes, provides a formalism for the description of processes, helps to guide the development of process flow languages and representations, and supports a number of process-related activities including process design and process control.

Part II

Representations for Process Design

Chapter 5

Process Flow Representation

The design and fabrication of an integrated circuit is a complicated endeavor. The single most important motive for a representation of the fabrication process is to enable such processing to occur – to tell someone or something how to make an integrated circuit. As the size of the process increases (number of unit process steps, mask levels), and the number of processes to be designed and executed within a single facility also increases, the use of information technology becomes essential. A second driving motive behind a computer-readable and computer-manipulable representation of the process, then, is to enable the use of such information technology. The manufacturing information system must be able to understand and access process information in order to guide operators or machines during fabrication, to schedule the operations within the facility, or to simulate and predict the effects of processing. A third motive guiding the process representation described in this chapter is to integrate these activities to better support the overall design and manufacturing endeavor.

The MIT PFR (Process Flow Representation) is a knowledge representation approach to process specification. Three key aspects of the process representation prob-

[†]The PFR is the original creation of M. McIlrath, and this chapter is drawn in part from collaborative work on the PFR reported in [11].

lem are addressed by the PFR: (1) conceptual models of processing, (2) a format for the interchange and transmission of process information, and (3) a program interface to enable the use and manipulation of process information by computer programs. Experience with the current implementation of the PFR has shown it to be helpful in supporting the demands of process design and execution, and indicates areas for future improvement of the representation.

5.1 Requirements

The need for a single, comprehensive representation of integrated circuit fabrication processes was first considered in detail by Ossher and Reid [35]. While the approaches chosen in Fable and in subsequent research differ substantially, all attempt to satisfy some subset of the goals and requirements set out in the Fable work. The fundamental requirements identified in [35] are completeness, readability, safety, portability, dynamic modifiability, and suitability for processing by other programs. Based on these fundamental requirements, a number of necessary features were also identified, including appropriate domain of discourse, abstraction mechanisms, verification and run-time checking, handling of equipment malfunctions, source-level interpretation, and suitable run-time support. These requirements and characteristics form the basis for evaluation and comparison of various approaches to process representation.

5.2 Existing Process Descriptions

Because the process is so central to semiconductor manufacturing and design tasks, it is not surprising that there exist many approaches to representing the process. These range from informal written documentation to formal descriptions serving as input to design or manufacturing support programs.

5.2.1 Written documentation

Perhaps the most common form of process representation is written documentation about the process. Ironically, written documentation often goes the greatest distance towards meeting all of the goals and requirements described earlier, with one exception: written documentation is not amenable to meaningful processing by computer programs. Written descriptions of the process can be arbitrarily extensive, and coupled with knowledgeable support staff who "execute" those descriptions can be argued to be complete, readable, inclusive of safety restrictions and warnings, portable (but limited by differences in background and contextual knowledge of its readers), modifiable, appropriate to many domains of discourse, supportive of many levels of abstraction (*e.g.*, through detailed operating procedure manuals and condensed run-sheets), susceptible to manual verification and run-time checking, responsive to equipment malfunctions, and suitable for human-driven run-time support of the fabrication facility. As the amount of written documentation increases, however, there are practical limits to the ability of humans to understand, manage, and use that information. Nevertheless, written documentation remains an essential (and in some cases, the only) representation of process information. Such written documentation might include: equipment manuals, equipment log books, facility operating procedures, unit process step procedures, design information (specifications, cross section pictures, *etc.*), individual process sequence descriptions, run-sheets for individual lots or wafers, and daily schedules for lot moves. Most of these descriptions are not strictly or solely process information, but rather contain information about both the process and other objects or activities within a facility. This has an important implication for the computer representation of processes and on the software support system required, and is discussed further in Section 5.5.3.

5.2.2 Computer-Accessible Information

A limited way to apply computer technology to the abundance of process information described above is simply to use the computer to help store and manage parts of that information. The most common example is to make parts of this information accessible *by computer*; that is, rather than a bookshelf and filing-cabinet repository for process information, computer files are used instead. This has several advantages, including the ability to more readily share and access information, and importantly to minimize the need for paper within the cleanroom. Utility programs may even be written to help locate, access, and modify process information. An approach that is often used is to generate "traveler" files containing the sequence of steps a lot is to undergo. Each step might be another file containing limited details of the process parameters, procedures, and perhaps desired measurements for that step. Copies of these files might be edited to record measurements and comments during processing for each lot. Even this degree of aid by the computer is possible only by imposing a limited *structure* on the representation of the process. An important generalization of this observation is that successively increased structure in process descriptions enables successively greater manipulation of process information by computer programs.

5.2.3 Computer-Manipulable Information

The limitation to the above approach is that process information is accessible *by the computer* but not *to the computer*. That is, computer programs are unable to understand or manipulate the semantic content of these files. For instance, it is very difficult to extract measurement data for engineering analysis from unstructured data files. If one imposes additional structure (ranging progressively from conventions on the use of "comment" fields, for example, to strict tabular or other organizations of data within files or databases), then it becomes possible to write programs that can locate and extract that information for further analysis or manipulation.

Conceptually, widely used work-in-process tracking systems such as COMETS [36]

and PROMIS [37] are extensions of the computerized traveler and run-sheets described above. In these cases, the structuring of information is more strict, data often resides in relational databases rather than text files, and the software support system is much more extensive.

5.2.4 Program-Specific Process Descriptions

Several examples of *specialized* languages or process descriptions exist in semiconductor processing. In each case, a representation has been developed for use by (usually a single) programs to support some activity domain. The WIP description of a process, for example, is usually sufficient to meet the needs of only one specific kind of activity: the management of actual processing. To describe the information necessary to perform scheduling, throughput, utilization, and other manufacturing analyses, several well known commercial [38] and research high-level simulation systems [39, 40] are being developed. Each of these typically includes its own description of the process. To support the process simulation activity, many different input languages have been developed to serve as input to specific process simulators (such as Suprem-III [24]). These process descriptions are generally incompatible and are not used by other than the specialized program.

5.3 The Dilemma: Multiple Descriptions

Many requirements remain unsatisfied by the existence of specialized and separate representations each designed to support a specific application of process information. First, there are problems within each specific activity domain. If a designer, technician, or engineer wishes to use more than one program in support of the same activity, it is necessary to learn more than one language. The translation of information from one input form to another (as illustrated in Figure 5.1) is usually done by hand and is a difficult and error-prone procedure. Both manual and automatic

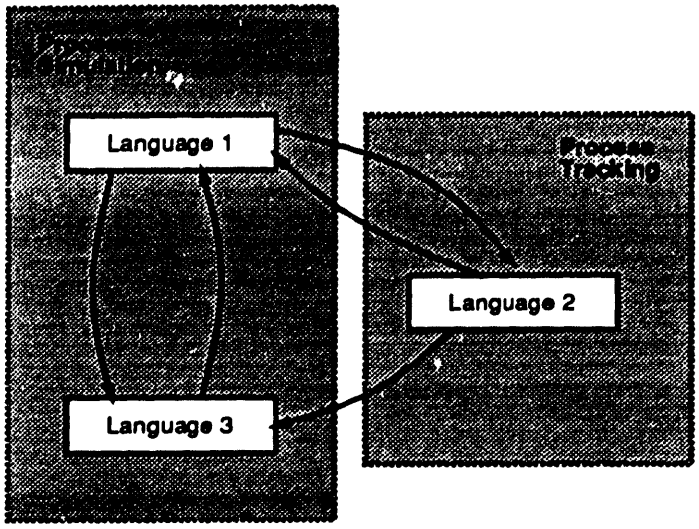


Figure 5.1: Multiple programs within a single activity domain (such as process simulation) and across domains (process simulation and work-in-progress tracking) typically require unique input languages. Automatic translations between these languages is difficult.

conversions are complicated by the fact that the information expressed by different languages overlaps but is not identical. For example, language 1 may be able to express photolithography steps, while language 3 may not.

Second, and more importantly, multiple descriptions pose a serious problem when one attempts to cross activity domains. Perhaps the most severe case is in transferring a process from the design phase to the manufacturing phase of its life cycle. This transfer now typically requires the generation of a process description that can be used to "run" the process, drawing from the description that was used to drive process design or process simulation. In the transfer of the process, information is often lost, and new information must be generated. After the transfer is complete, it is often difficult to "transfer back" the process for engineering analysis because of lost information. It thus becomes extremely difficult to ensure that one is fabricating what one has designed, or that one is simulating what one is making.

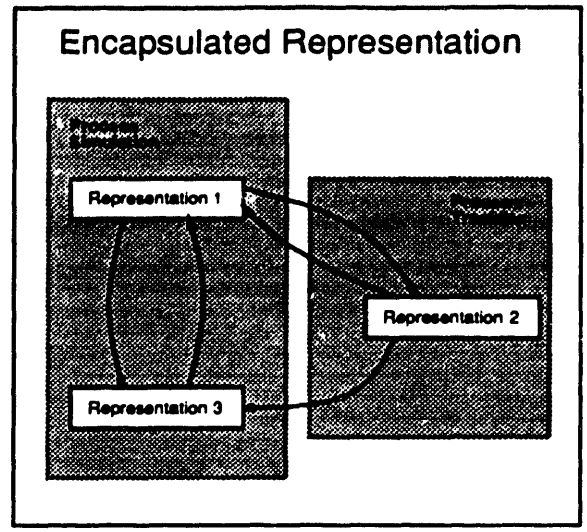


Figure 5.2: Encapsulation of separate input languages within an "umbrella" process representation. Problems of data sharing and exchange remain.

5.4 The Solution: A Unified PFR

The approach taken in this work, and in concurrent process representation research elsewhere [7, 29, 35], is to unify both process information and process activities using a shared common language or representation. The MIT Process Flow Representation (PFR) supports the unification, as opposed to encapsulation, of process data and activities.

In *encapsulation* multiple input languages are joined together under an "umbrella" representation which embeds within it the multiple languages needed for different programs and activities. This approach helps address two problems. First, it becomes possible to associate one process description with a parallel description of the same process or process step as needed by another program. Second, the ability to physically transfer process information between different locations is enhanced. Encapsulation does not, however, address the problems of sharing information between different activities or programs, nor does it ease the problem of translating between different descriptions. As shown in Figure 5.2, encapsulation is only a small improvement over

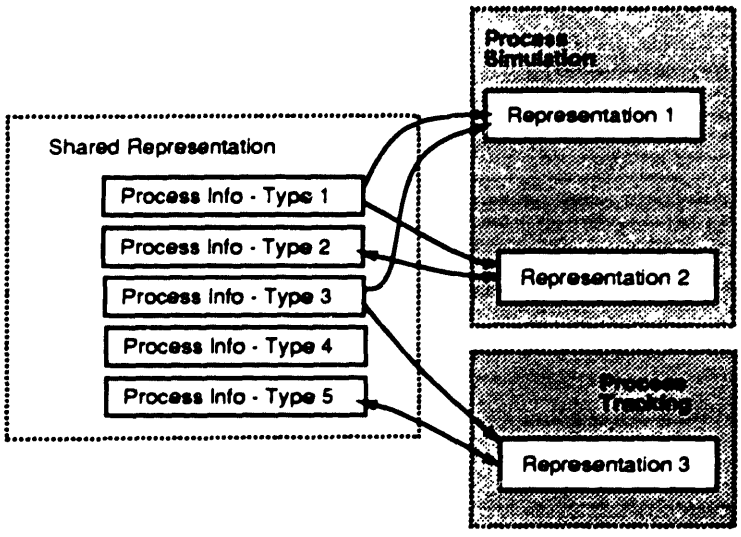


Figure 5.3: Sharing of process information by representation in a program-neutral format.

the multiple description dilemma.

An alternative approach whereby information is *shared* is illustrated in Figure 5.3. Here, every effort is made to represent the essence of the process information in a *neutral* and unique fashion. Each program within the same or different activity domains then has access (via translation or via direct interfaces) to the subset of process information it needs to perform its activity. Only one translation or data filter for each program is necessary, and information about the process can be shared among all of the activities involved in the design, use, and support of the process. This suggests a fundamental principle behind the PFR:

- Represent the basic process *information*, and not the *input* needed for different activities.

Unification of process representation should be via the union of the semantic content of process information across multiple activity domains rather than via the incorporation of syntactically different descriptions used by different programs or activities.

A second principle also guides the development of the PFR:

- Enable fine-grained correspondences among process data.

Process information must be broken into components that relate to each other in established ways in order to achieve a unified process representation. To understand this requirement, consider a shared representation where the sequence of machines for the entire process is listed in one location, and where a sequence of wafer descriptions for the entire process is listed in another. The relationship between the two is only captured at a very coarse level: the entire machine sequence somehow "goes with" the entire wafer description sequence. More finely-grained relationships and correspondence of information are necessary to bind together individual items of different information types. Such correspondence could be sequential (*i.e.* a one-to-one correspondence). Most of the recent process description approaches use a hierarchical correspondence.

5.5 PFR Conceptual Model

5.5.1 Decomposition and Association

A *process flow* consists of some number of *operations* which are performed on *wafers*. An operation may consist of a time-ordered sequence of any number of component operations (or *sub-operations*), leading to a hierarchical decomposition of process information. This decomposition can be pictured as a tree (as in the example process flow shown in Figure 5.4), where the decomposition proceeds from top to bottom, and where the sub-operation sequence for an operation is a left to right ordering of the children of that operation. Process trees are also often depicted with decomposition from left to right, and time sequencing of children from top to bottom (as shown in Figure 5.5).

The second fundamental feature of the PFR is the association of *attributes* with operations. An attribute is some piece of data that applies to an operation *as a whole*. In the process tree of Figure 5.6, for example, we see the association of the time-required to complete an operation with each node, including interior nodes, in

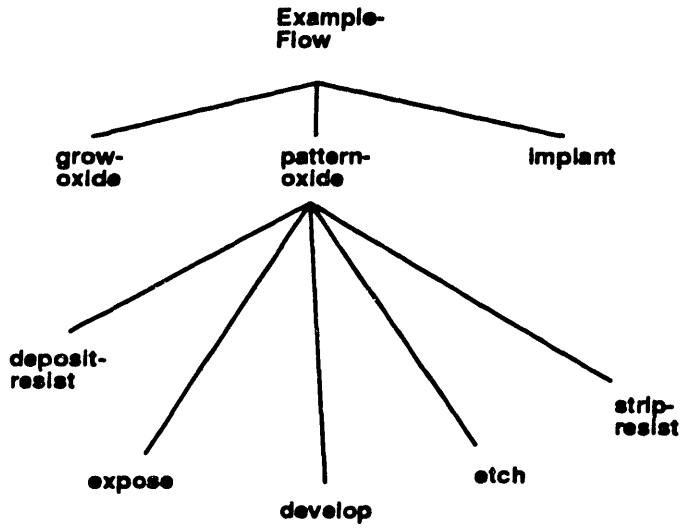


Figure 5.4: Hierarchical operation decomposition in the PFR.

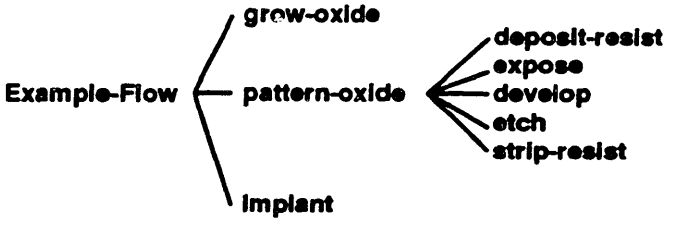


Figure 5.5: Alternative tree showing hierarchical operation decomposition in the PFR.

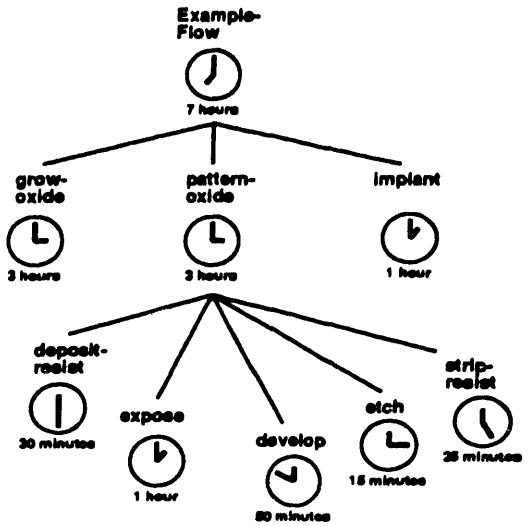


Figure 5.6: Example showing association of time-required attributes with operations in a process tree.

the tree.

Together, these two basic mechanisms – operation decomposition and attribute association – enable the description of arbitrarily complex process steps. The determination of what attributes are necessary, and how such attributes should be structured, is at the heart of process representation research.

5.5.2 PFR Attributes

In order to achieve the sharing of process information as discussed earlier, application-neutral representations of attribute information is critical. The PFR consists of a small number of named and predefined attributes denoting specific categories of information. These attributes are closely based on conceptual modeling of the structure and nature of semiconductor fabrication. The generic process model discussed in Chapter 4 provides the fundamental theoretical basis for several of the attributes in the PFR. These include the `:change-wafer-state`, `:treatment`, `:settings`, and `:readings` attributes of the PFR. Other attributes provide general information about the oper-

ation for use in other activities, including scheduling and process execution. These include the `:time-required`, `:permissible-delay`, `:machine`, `:instructions`, and `:documentation`. The structure and semantics of these attributes can be found in the PFR User's Guide (Appendix A).

5.5.3 PFR Objects

Operation attributes are defined as data associated with operations. An attribute may consist of or contain other objects and attributes, depending on the allowable structure of that attribute. For example, the `:body` attribute of an operation contains the sequence of sub-operations which together make up the operation. As mentioned in Section 5.2.1, representation of and reference to many objects in addition to operations may be necessary in the description of a fabrication process. Description or access to the following objects is provided by the MIT PFR.

Flow and Operation

An *operation* is the hierarchical unit of processing. A *flow* is identical to an operation, but usually contains two or more sub-operations.

Values

A *value* may be an enumerated data item or a quantity. Quantities may be either dimensionless or include units. Examples of such values in the PFR include time, distance, temperature, and numerical values. Only a limited set of units are now supported by the PFR. All quantities are converted to quantities in a base unit for the data type. Once they have been so converted, no units information remains – the only type information remaining is integer, float, *etc.* A more comprehensive approach has been implemented in BPFL by Williams [41].

Many of the specifications and parameters used in a process description are not known exactly, so that *inexact values* are important. One type of inexact value

is a simple *interval*, where a numerical value may lie anywhere within an upper and a lower limit. Additional information about the probability of a value is also possible. Uniform, Gaussian, or other probability distributions might be defined on some interval. In the PFR, a value is an object which may have units, a mean, and positive and negative ranges from the mean. The positive and negative ranges define an interval containing the mean, but do not currently imply additional statistical or probabilistic information. The numerical manipulation of values is complicated by inexact values, particularly in those cases where the set of inexact values (*e.g.*, Gaussian distributions) is not closed over such operators as addition or multiplication. In the PFR, inexact values are manipulated as mathematical intervals, and means are manipulated as single valued numbers.

Materials

During processing, material is added, removed, converted, and consumed near the surface of a wafer. Material information is presently handled in an ad-hoc fashion within the PFR. The definition, representation, and manipulation of material and material properties is an important part of work toward a microelectromechanical CAD system [42], and the description of materials in PFR can be expected to benefit from that work.

Masks

Information about the mask, including layout layers and their transformation into a mask (particularly whether the mask is clear or dark field), is necessary both for processing and for simulation. Specific mask information, such as the name and location of the physical mask itself, may be needed during processing. Generic mask specification via geometric combinations of layout layers is described in Chapter 7.

Cross Sections

Logically, a cross section defines locations on the wafer which will see certain kinds of processing. This is particularly useful for specifying areas of interest for simulation (see Chapter 7); its usefulness in modular process specification is less clear.

Wafer

The exact representation of a wafer will depend on the kinds of tools that are manipulating the wafer or wafer model. For fabrication, it may be necessary to know the lot that the wafer belongs to, as well as the collection of measurements made to the wafer. Information about the *wafer state* as it passes through a process must be maintained. The development of a Profile Interchange Format (PIF) for wafer representation is described in Chapter 6.

Equipment

Several PFR attributes provide for interaction with fabrication equipment, including the *settings*, *readings*, *machine*, and *instructions* attributes. The best interface between equipment and the PFR remains to be determined. Currently, names of machines (or sets of machines from which one can choose) may be specified in the PFR. Extensive equipment information (best organized via some class hierarchy) should be visible and available to process applications. For example, a fabrication interpreter may need to query the equipment state before allowing processing to proceed.

Just as the description of operations should be unified, so too should there be a unified representation of these objects to facilitate sharing and integration of process-related software among other programs within a comprehensive manufacturing information system. At the same time, a difficult question is to decide what is and what is not part of the PFR itself. If something is excluded from the PFR, it is necessary to define interfaces with external representations or objects. In particular, interfaces to standard wafer representation (not yet supported in the PFR) may involve queries

and assertions about wafer state.

5.6 Interchange Format

An interchange format for process descriptions is necessary to meet a number of requirements. A textually oriented (computer readable) interchange format helps to facilitate the transmission of process information between different programs, between different sites, and between different factory support systems. A textual format provides the basis for the implementation of PFR interpreters in different programming languages (such as Common Lisp or C++) and on different databases or object systems. It is also desirable that the interchange format be simple enough that interpreters can be embedded within stand-alone tools that wish to support a standard interface to process information. In particular, the textual PFR might provide process information directly to new process simulators (such as those for the modeling of mechanical structures and properties [42, 43]).

The PFR can be thought of as a knowledge representation language [11]. It provides textual mechanisms for describing the decomposition of a process into small operations and for the association of attributes with each of those operations. The syntax is Lisp or Scheme-like. A small number of language constructs are provided: procedure definition mechanisms support abstraction and parameterization, and an *if* construct supports branching and control flow. A set of basic numeric functions are also supported. The textual format (in addition to the semantics) of the MIT PFR is described in detail in Appendix A.

5.7 Program Interface

In addition to an interchange format, a powerful and convenient programmatic interface to process information is essential. If the program interface can also be made standard across a broad range of implementations, then the integration, development,

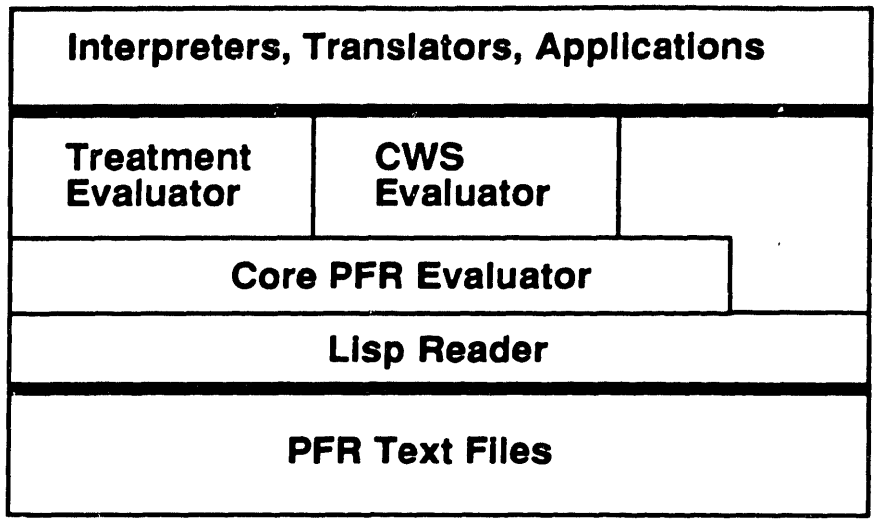


Figure 5.7: The structure of the PFR support system.

and exchange of tools based on that interface is further simplified.

In this work, a Common Lisp interface to the PFR originally developed by McIlrath has been adopted and extended. The essential architecture and layering of PFR program interfaces for a typical application is shown in Figure 5.7. This figure is based on the interaction of the Simulation Manager (Chapter 7) with the PFR; other applications and tools may use only some of these components and may require additional interfaces (*e.g.*, a scheduling interpreter might only need the core evaluator, while a fabrication system might also require `:instruction`, `:machine`, and `:settings` evaluators).

First, the Common Lisp reader is used to read and evaluate text files containing PFR descriptions; this builds a symbol table relating definition names with definition values and functions. Second, the core evaluator is responsible for converting uninterpreted forms (such as the bodies of definitions) into CLOS objects that application programs may manipulate. These CLOS objects may reside in program memory, or may reside in a shared object-oriented database (Gestalt), so as to provide persistence and shared access by multiple programs [44]. The essential role of the core evaluator

is to "squeeze out" the language-dependent aspects of the PFR: parameters are substituted in procedure bodies, constants are substituted, conditionals evaluated, *etc.* Third, adjunct evaluators with syntactic knowledge of the internals of different PFR attributes evaluate these attributes and return additional objects. For example, a *treatment evaluator* knows about the possible forms within the `:treatment` attribute of a PFR operation, and returns a treatment object wh ı the application program can pick apart using CLOS access methods. On top of these basic evaluators, application specific interpreters are constructed that access a process description, calling the core and layered evaluators as necessary.

A third principle guiding the PFR has contributed to the separation between interpreters and PFR descriptions:

- Keep the process flow representation as simple as possible.

The PFR strongly emphasizes the declarative specification of information about a process, and minimizes the manipulative and computational flexibility accessible within the PFR. Interpreters, each customized to support some particular application domain or tool, have the responsibility for the manipulation of PFR objects. The full power of existing programming languages, then, is used only within the interpreter and not within the PFR itself. In particular, *methods* (generic functions with different implementations for different argument types) are supported only within an interpreter, and not within the PFR itself (where only simple functions are supported). This approach is in sharp contrast to that of some other process representations, where the user specifies operation data objects and methods.

5.8 PFR-Based Tools

A number of PFR-based tools have been developed, some by the CAF project and some as part of this thesis. Tools contributed by this thesis include a PFR syntax checker, a process flow *Simulation Manager* (described in Chapter 7), a limited de-

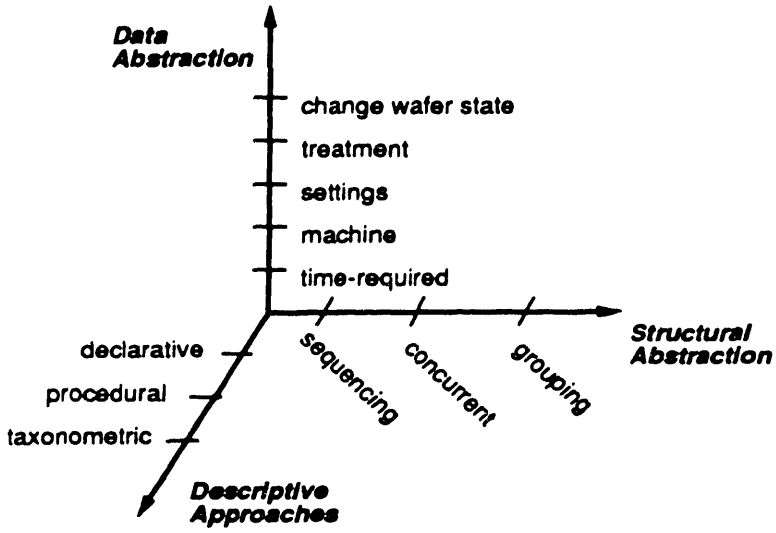


Figure 5.8: Conceptual matrix of process representation.

sign rule checker (Chapter 10), and a traveler or run-sheet generator (Chapter 11). Additional tools developed by workers in the CAF project include a process flow editor [14] and a fabrication interface [15].

5.9 Discussion: A Matrix of Possibilities

A fundamentally confusing aspect of the process representation problem is that there are many possibilities for descriptive and implementation approaches (as summarized in Figure 5.8). Along one axis in the matrix is structural abstraction, along another lies data abstraction, and along the third axis lie descriptive approaches. There are many possible ways to describe a fabrication process, each one occupying some subspace within the three-dimensional matrix formed by these axes.

The *data abstraction* axis is the simplest in general, but most difficult in its specifics. All process representations strive to enable expression of various kinds of process information. Deciding exactly what attributes are needed, and the structure of those attributes, must be made based on conceptual models of process informa-

tion, and are guided by the needs of the activities to be supported. While a great deal of process representation research has so far focused on general data, structural, or descriptive abstractions, additional research is required to explore the attribute dimension.

The *structural abstraction* direction is fairly straightforward. An operation can be broken up into smaller operations, which can then be sequenced in time. These sequences may be hierarchically organized, as in the PFR. Parts of a process description may also be broken into smaller parts which are grouped in some other fashion. For example, elements within a `:change-wafer-state` attribute may be grouped together to imply that several changes are all occurring simultaneously rather than sequentially. The description of processing in which multiple actions are to be coordinated may require expression of parallelism and concurrency.

The *descriptive mechanisms* expressed along the third axis are more difficult to understand. Attributes that apply to an operation might be expressed declaratively, procedurally, or via classification and inheritance mechanisms. An operation can be viewed much like a procedure in a programming language, and attributes can then be expressed as arguments. Rather than being arbitrarily definable, a declarative approach imposes a structure on (and provides a place to put) the things one has to say about the operation. This is an orthogonal axis to that of structural decomposition; it is possible to decompose without calling procedures, and similarly it is possible to define and use different descriptive approaches without necessarily decomposing one or more attributes. A third descriptive mechanism is taxonomy (and inheritance), where process information is received from more general classes or instances of process objects rather than via explicit procedural parameters or declarative attribute attachment. Again, possible variants of taxonometric classification and inheritance mechanisms are possible when combined with the other axes of the representation problem. In the PDA [30], for example, attributes can be inherited along both "is-a" and "part-of" hierarchies.

The representation of fabrication processes is crucial within both technology CAD systems and emerging CIM systems. Research into the representation of processes has been active over the last several years, and each supports some subset of the space defined by these axes. Fable was the first attempt to develop a comprehensive description of the process [35], and took a programming language approach to solving the problem. Data groups were identified and handled as procedural arguments. Decomposition along multiple hierarchies was proposed, with an emphasis on time sequential decomposition. A key contribution in Fable was recognition of the need to specify information at multiple levels of abstraction.

The Berkeley Process Flow Language (BPFL) [29] has continued in the theme of a procedural, language-based representation approach. Arguing that the kinds of attributes needed cannot be known *a priori*, the BPFL has opted to express the information suitable for specific applications in multiple "views" of the process. The issue of views versus attributes is discussed further in Section 5.10. A central goal of BPFL is the control of manufacturing, leading to an emphasis on the procedural description of processing.

MKS ("Manufacturing Knowledge System") is a comprehensive IC CIM system under development at Stanford [7]. A key component of this system is the representation of process information. An extensive exploration of the use of process knowledge for simulation has been performed by Wenstrand [30]. The "Process Design Aid" (PDA) provides mechanisms for the definition, manipulation, and use of process descriptions. A process step is described as a *process object*, which includes information about process parameters and process simulator inputs. Other related objects include *device goals* summarizing the goals and requirements of some process. PDA makes extensive use of a prototype-instance object model provided by HyperClass [45]. In the prototype-instance model (another example of which is the KR system by CMU [46]), each instance itself forms a new class which can be further specialized. This is in distinction to class-instance models where classes can be refined,

but instances are based only on class definitions. The prototype-instance paradigm is adopted in PDA as the basic mechanism for the definition and specialization of process objects. Inheritance occurs not only via "is-a" hierarchies, but also along the "part-of" (and perhaps other) hierarchies. The definition of the structure of attributes is generally left to the user (though some attributes are predefined). The PDA has as its central goal the representation of processes to support the simulation and enhancement of existing processes. As a result, its emphasis is on mechanisms for the bottom-up characterization and use of process steps, particularly via tuned simulator models and descriptions.

The PFR explores the declarative regions of the process specification possibility space. Procedural mechanisms are used to accomplish sharing of information and coupling between attributes. To emphasize the declarative nature of the PFR, the `define` statement declares equivalences rather than assigns values to variables. Thus

```
(define x y)
(define y 2)
```

results in

```
x => 2.
```

Evaluation of the body of a definition is deferred until the defined symbol is used (that is, until some interpreter requests the value). The user states what `x` and `y` *are* rather than an algorithm for computation. The danger of this approach is that definitions such as

```
(define a b)
(define b a)
```

may not result in successful evaluation [47]. The advantage is that the order of definition loading or access (from a database) does not matter.

In the PFR it is possible to decompose not only whole operations, but also to decompose particular attributes. Usually this corresponds to sequencing (in time);

for example, a treatment may be broken down into smaller treatments that apply for some portion of the entire treatment time. In other cases, the decomposition carries different semantics. For example, multiple change wafer state descriptions within a CWS attribute imply multiple changes to the wafer that occur cumulatively, and may happen either sequentially or simultaneously. In both of these cases, the same procedure definition and application mechanisms may be used to describe this decomposition. The real core of the PFR, however, lies in the adoption of a set of attributes (based in large part on a generic model of processing) for the declarative description of an operation. The PFR is intended to support not only process simulation and fabrication, but to more broadly support the variety of tasks required in process design. The essentially declarative nature of the PFR is thus neutral to the different interpreters or tools that access or build up knowledge about a process.

5.10 Attributes vs. Views

There is an important distinction in the PFR between *views* of the process and *attributes* of the process. An attribute is a basic piece of information about the process, while a view may be a collection or bundling of several attributes that are needed to support some kind of activity. A view might also correspond to the way that different classes of users or people think about the process. Some of these views, and the attributes that they make use of, are summarized in Figure 5.9. The distinction between views and attributes is important, because it emphasizes the fact that there is a great deal of overlap of information between different views of the process. Implementations that maintain a separation between these descriptions during process interpretation, for example, would be difficult to use (a single attribute hierarchy is usually insufficient to the needs of one interpretation). The PFR uses attributes rather than views to describe a process, and does not provide any formal support for views. Instead, each interpreter is free to define for itself the collection of

View	Attributes
generic process model	change wafer state, treatment, settings
scheduling	time-required, permissible-delay, machine
fabrication	machine, settings, instructions, readings
documentation	doc, version
design	change wafer state, treatment, settings, machine

Figure 5.9: Views as collections of PFR attributes.

attributes it wishes to access or manipulate.

5.11 Grids and Layered Graphs

The PFR bears some resemblance to the "grids" and "layered graphs" proposed by Ossher to support the Fable process flow language [48]. The purpose of an *abstraction grid* is "to provide flexible, concise, and readable structuring mechanisms." As shown in Figure 5.10, a grid consists of two dimensions. First, the different *views* in a program or representation are shown as horizontal planes. Second, an individual object can contain several views; each object is shown as a vertical plane. The third dimension is for pictorial convenience only. A node in the grid defines a particular view within a particular object. A relationship between two nodes (a reference, function invocation, *etc.*) is shown as an arrow between the nodes. Each horizontal plane and the connected nodes within a view is called a *group*, and defines the relationship between different objects at the same level of abstraction. While a goal of the grid mechanism is to concisely and clearly define the relationships between different nodes, in practice the full flexibility provided by the grid makes it difficult to write and to conceptually understand multi-view programs (including Fable programs). The complicating condition is that any node can relate to any other node in the grid, even

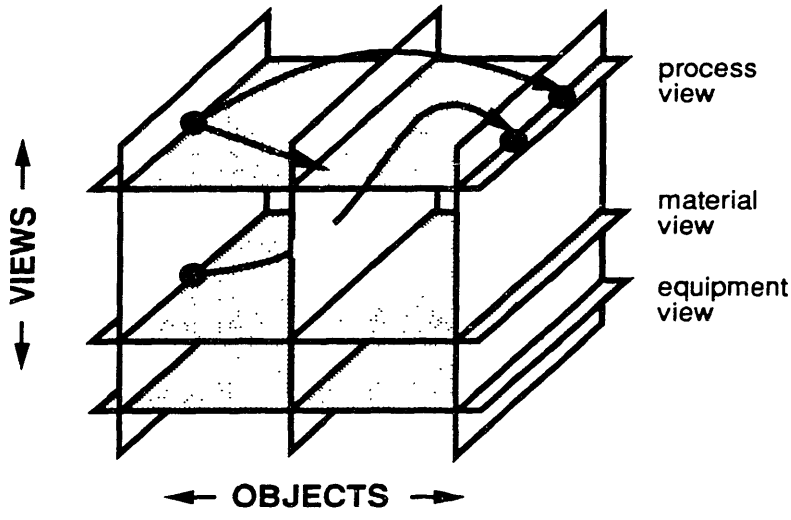


Figure 5.10: Ossher's "grid" structuring mechanism [48].

across group boundaries.

The PFR, because it uses multiple abstraction layers or attributes, resembles the grid. However, the flexibility of the grid has been restricted or simplified in a crucial way, as illustrated in Figure 5.11. First, the unit of description is an *attribute* rather than a *view*. Within any attribute, one writes specifications or provides information using only valid primitives of that particular attribute. The `:body` of an operation, for example, can only refer to other operations. Once inside the `:treatment` attribute one can only specify (directly or via named and potentially parameterized definitions) information via valid primitives of the `:treatment`. One cannot, for example, invoke an operation from within the `:treatment`. The hierarchy of (sequential) operation decomposition is the basic scaffolding on which relationships among different attributes hang. An application program (or interpreter) which wishes to draw information from a set of different attributes can find such information by traversal of the operation hierarchy.

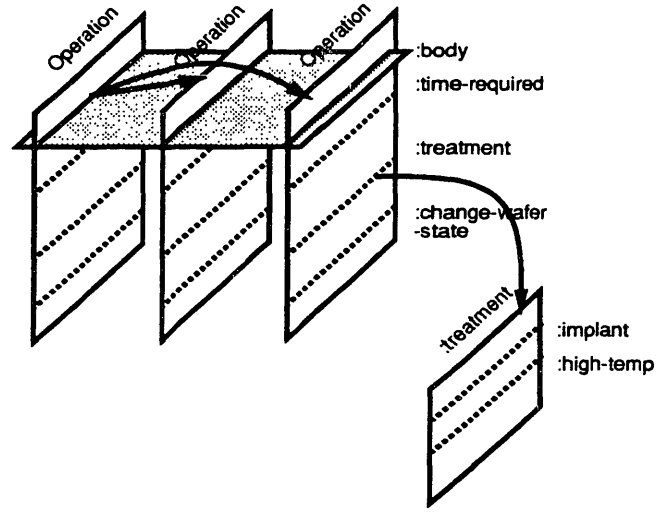


Figure 5.11: PFR simplification of the layered graph.

5.12 Attribute Consistency

Another important characteristic of the PFR is that while it defines attributes for expression of information, it does not mandate the relationships between these attributes. First, *consistency* between attributes is not enforced by the PFR. The PFR provides mechanisms to organize and express information, but it is the duty of external programs (*e.g.*, design rule checkers) to impose or verify consistency between these descriptions. For example, it is not necessary that the `:treatment` and `:change-wafer-state` attributes be consistent. Furthermore, the PFR does *not* assume that some attributes are *specifications* with respect to other attributes that must be *implementations* of those specifications. Thus, it is possible to interpret the `:change-wafer-state` as a specification of what a step is "supposed" to do and the `:treatment` as a derived implementation of that desired change. It is also possible, on the other hand, to consider the `:treatment` as the specification of how the step is performed, and the `:change-wafer-state` as a derived *characterization* of that treatment. From a practical standpoint, it is important that the PFR itself not be saddled with consistency requirements so that it can serve as an active, living data structure dur-

ing the design of a process (when incomplete and inconsistent descriptions abound). Rather, the PFR should be thought of as a central repository for process knowledge which will grow and change during the design, analysis, or execution of the process.

5.13 Execution of Specifications

A goal of process representation is to support the implementation of application specific processes. An extreme extension of this goal would be to automatically generate processes given a specification of desired electrical or mechanical properties, a final wafer structure, a particular change in wafer state, or some other "high-level" or incomplete description of a process. The automatic generation of a process description is closely related to similar problems in the programming language, robot planning, and artificial intelligence domains.

In programming languages, there has historically been a great deal of interest in what has variously been termed nonprocedural, "very high level," or specification languages. As pointed out by Leavenworth and Sammet in an early overview on the subject [47], the central idea behind such languages is to enable the user to describe *what* he or she wishes to achieve rather than *how* the solution is achieved. An important observation in that same paper is that such notions are *relative* to available implementation technology. Today's specification language is tomorrow's target language, and today's "automatic program generator" is tomorrow's compiler. Nevertheless, the trend is toward declarative, functional, or other language approaches which show certain characteristics, including associative referencing, aggregate operators, elimination of arbitrary sequencing, or nondeterministic programming.

The separation of specification and implementation is an important idea behind such structured programming languages as CLU [49]. As an example of specification execution approaches, Zippel proposed to enforce separation such that module invocation could only be achieved via matching specifications and not by named pro-

cedures [50]. An interesting conclusion of that work was that as specifications move out of comments, the programmer is forced to make the specification more precise, and that this precision enables a programming system to aid in the development of large programs.

A large part of artificial intelligence research and practice could be described as "automatic problem solving", and expert system, knowledge representation, and other AI approaches may also be applicable to fabrication process generation and implementation. In very complex control and automatic planning systems (*e.g.*, to support robotics), a common problem is to develop a *plan* to achieve some *goal* [51]. The execution of specifications (or the generation of plans or processes) generally requires, first, a (relatively) nonprocedural description of the goals, and second, the expression, use, and manipulation of *procedural knowledge* [51]. Approaches to the representation of procedural knowledge generally involve the description of state information (the *world state*) and changes (or sequences of changes) in the world state via *actions* or *events* [51]. Such information is very similar to that emphasized in the generic semiconductor process model of Chapter 4, and automatic planning research is an exciting area for exploration and potential application to semiconductor process design.

5.14 Modifications to the PFR

The PFR can be improved in several ways. First, the PFR should be expanded to more completely support the generic process model described in Chapter 4. An expressive description of treatment information is needed (perhaps via an environmental parameter "waveform" language), so that that a clean separation between machine/settings, treatment, and change in wafer state descriptions can be maintained. Mechanisms for the description and management of models (as defined in the generic process model) should also be investigated. Second, there need to be mecha-

nisms for the specification and query of wafer state descriptions. These should enable specification of starting material, and specification of invariants or requirements on the wafer state at the end of a step. Finally, inheritance mechanisms (perhaps similar to those used in PDA) should be investigated, so that slightly modified steps can share the descriptions of common parts without requiring modification of previous uses of the step (as often is necessary in the procedural implementation).

5.15 Summary

A process flow representation is crucial to support the design and execution of application specific or custom fabrication processes. The PFR described here is fundamentally a knowledge representation which helps to unify the information and activities involved in semiconductor design and manufacturing. The process representation has three aspects: (1) conceptual models to inform decisions as to what kind of information should be maintained by the PFR, (2) an interchange format for the definition and exchange of information between tools, sites, and systems, and (3) program interfaces so that interpreters and other tools may access, use, and manipulate process information.

Chapter 6

Profile Interchange Format

Contemporary process and device simulators, and other technology CAD (TCAD) tools impose particularly difficult demands on wafer structure and device representation, data exchange, and tool integration. Not only is such information complex and voluminous, but TCAD tools often have different internal data structures and storage formats, complicating the exchange of information between tools. Work to improve this exchange has tended to focus either on common file formats for loosely coupling separate tools together [53], or on common data structures that tightly couple subroutines or programs together into a larger system [54].

As suggested in Figure 6.1, a conceptual model of microfabricated structure and device information (or *profiles*) can help to guide and make consistent these two facets of data exchange. First, the information can be expressed in a neutral file format, such as the *intersite* profile interchange format (PIF) [53], so as to enable exchange between tools at the same or different sites, and across different hardware environments. Secondly, the conceptual model can also guide generation of a programmatic interface giving TCAD tools direct access and manipulative power over profile data.

---

†This chapter is drawn from collaborative work on the PIF reported as "The Intertool Profile Interchange Format: An Object-Oriented Approach", D. S. Boning, M. L. Heytens, and A. S. Wong [52].

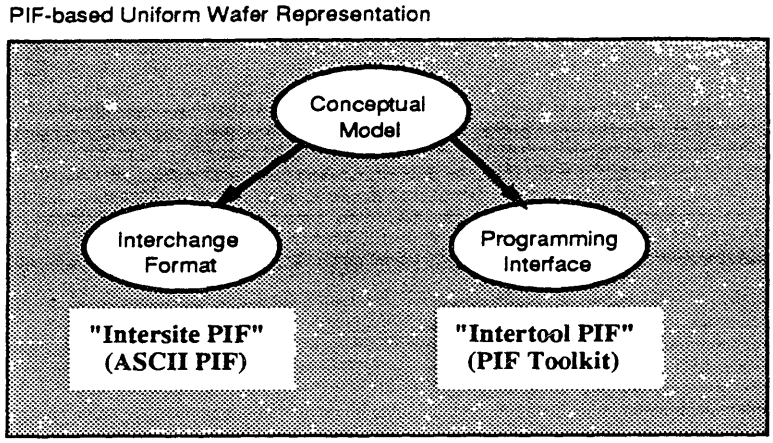


Figure 6.1: Uniform wafer representation using the profile interchange format requires both file format and programming interface versions. The first facilitates the exchange of information between tools and sites, and the second enhances the integration of TCAD tools.

The contributions of this chapter are twofold. First, a formal object-oriented approach is presented that helps in the definition and understanding of a PIF toolkit. Second, a programmatic interface, implemented and tested in PIF/Gestalt, is presented that enables the exchange of profile data and the integration of technology CAD tools in a consistent and uniform way.

Historical barriers to TCAD data integration are overcome by the intertool PIF, as discussed in Section 6.1. A formal wafer and device object model that corresponds to the intersite PIF is proposed in Section 6.2. A mapping from this object model to a set of functions forming an intertool PIF program library (or *PIF toolkit*) is described in Section 6.3, and examples illustrating the use of these toolkit functions are shown. The object model, function generation rules, and resulting toolkit are tested with the PIF/Gestalt implementation, described in Section 6.4. Based on the Gestalt object-oriented database (which has been developed in part for this work), PIF/Gestalt adheres to the object model by automatically generating the toolkit library from the formal PIF object definitions. Section 6.5 presents application examples demonstrating the use of PIF/Gestalt with existing TCAD tools (Suprem-III

and Suprem-IV), and in development of new tools (a PIF plotting program and an intersite PIF reader/writer). Finally, experiments to investigate the relationship between the intersite and intertool PIF are described in Section 6.6.

6.1 TCAD Data Integration

Problems in the exchange of wafer structure and device, or cross-sectional profile information between isolated TCAD tools have long been recognized [55, 53]. TCAD data exchange has focused on common file-format or on common program data structure approaches. In this section it is argued that a unified approach to data storage, data access, and data structuring is necessary for effective integration of both TCAD data and TCAD tools.

6.1.1 File Format Approaches

Technology CAD programs, including process and device simulators, parameter extractors, and graphical post-processors, have traditionally been written with incompatible internal data structures and data file formats. The communication of information between such tools requires interface or translation code, both external and internal, to each individual tool as illustrated in Figure 6.2. This combinatorial connection of tools becomes onerous as the number of tools grows. The establishment of uniform file formats, such as the intersite PIF [53, 56], a geometric data interface proposed by Kato [57], and the "device interchange format" [58], reduces the task of data translation, as pictured in Figure 6.3. Similar efforts and approaches have been developed in other engineering disciplines, including solid modeling [59] and electronic CAD [60]. A file-based format, however, continues to suffer from two limitations. First, each tool must still provide a neutral file reader and writer, or a separate translator between each tool-specific file format and the neutral format must be implemented. Secondly, while the file format provides some limited guidance in

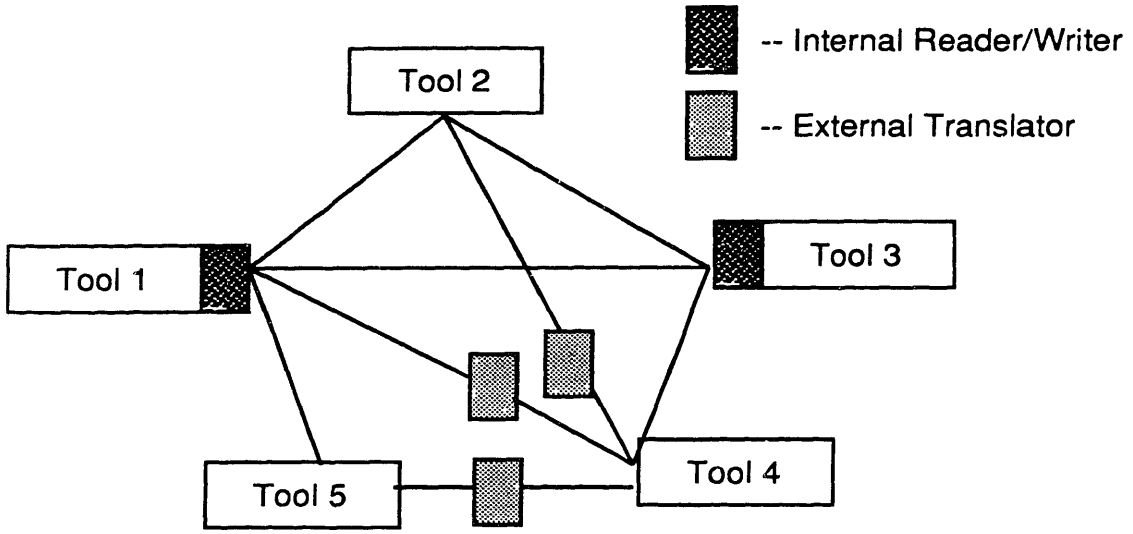


Figure 6.2: Exchange of profile data between tools using internal and external translators between each and every tool that is linked.

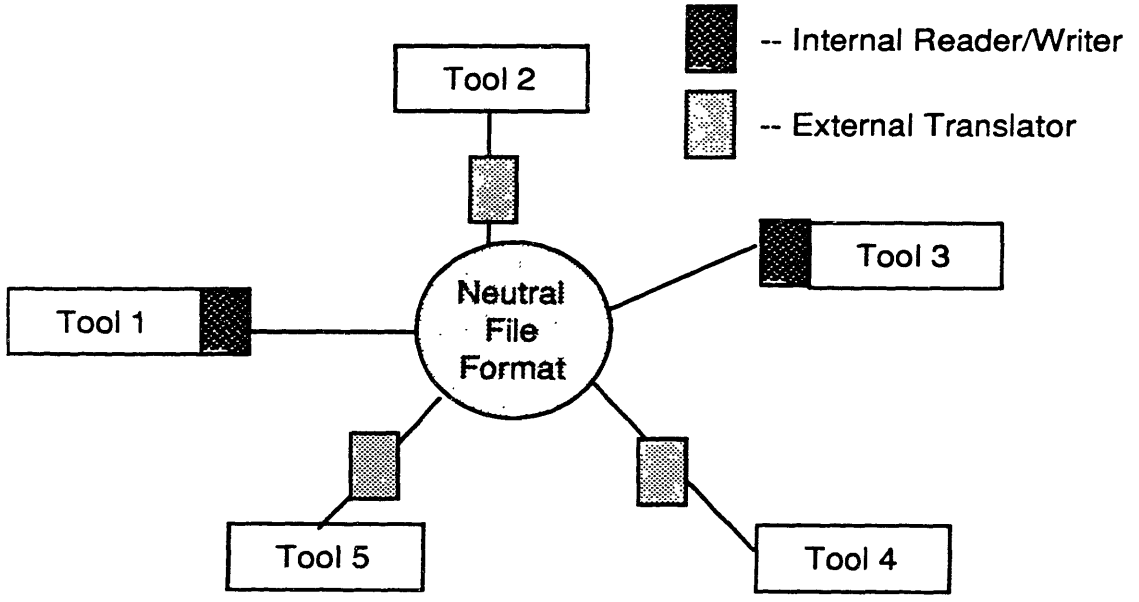


Figure 6.3: Exchange of profile data between tools via a neutral file format.

the organization of internal tool data structures, it does not provide mechanisms to manipulate such data structures. The file format eases the number of data interfaces that are needed, but does not help in the generation of data integration code.

6.1.2 Storage Access Approaches

One approach to address the reader/writer problem is to provide a set of I/O functions that facilitate access to files of profile data. SNC [61], for example, provides functions that directly manipulate memory-mapped data files containing profile data. These functions give TCAD tools direct or "random-access" to hierarchically organized binary data. Experience in attempting to interface tools using SNC, however, showed that these functions, which are generally of the form `snc_write_2d_float_array(array_name, array_size, array)`, limit applications to a "low-level" view of wafer data as named collections of strings, floats, arrays, etc. Data structuring and encapsulation of data collections into "objects" are not supported directly by SNC functions. Efficient storage and data access approaches such as SNC or SPIFI [62] enhance the ability to store and access data, but do not by themselves define a consistent conceptual view of that data.

6.1.3 Data Structuring Approaches

An alternative approach to the integration of TCAD modules and tools is to provide common data structures for use by TCAD programs. This approach can also ease the difficulty of persistent data storage and retrieval between executions of programs. In the CNET approach [54], for example, this corresponds to a straight-forward write or restore of Fortran-based data structures. The approach reported by CNET, however, imposes restrictions on the coupling of tools: TCAD modules must be linked together into a single executable program, or programs must be written in the same programming language to make use of language-dependent data structures. This is difficult given the heterogeneity of existing TCAD tools, and the growing variety of

programming languages used in these tools.

6.1.4 A Unified Approach

A unified approach to TCAD data integration, including data storage, data access, and data structuring aspects is now presented. The intertool PIF, with its implementation in a *PIF toolkit*, is illustrated in Figure 6.4. First, the *storage* of wafer

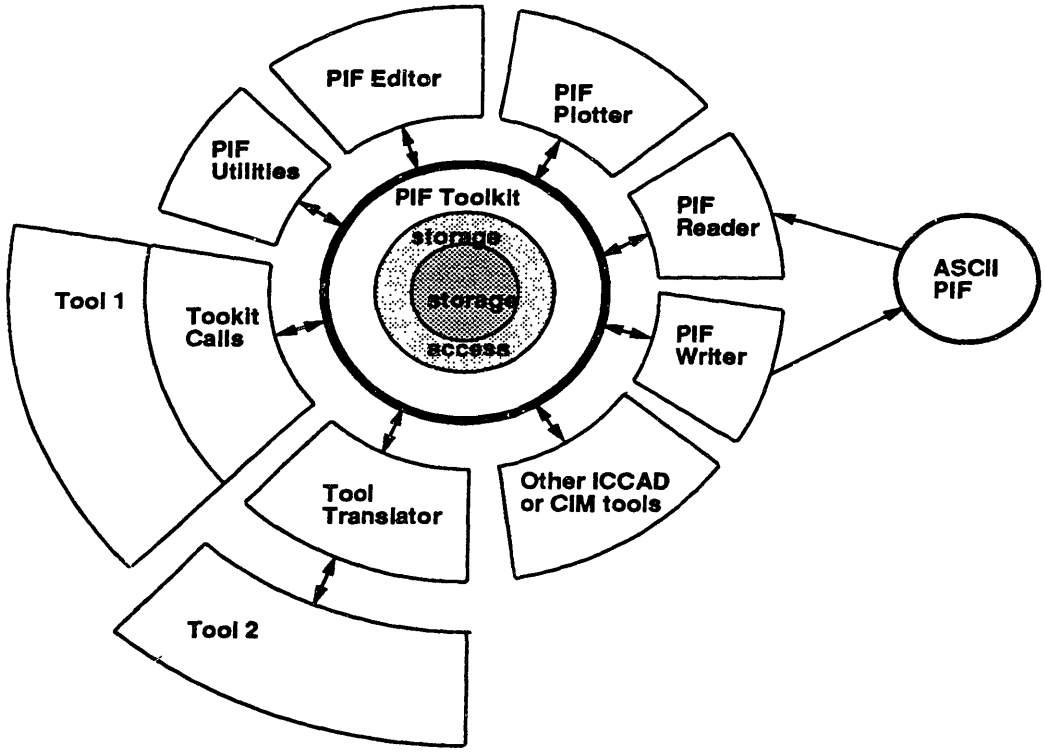


Figure 6.4: Exchange of data and integration of tools via a common programmatic interface to profile data.

data, as in Figure 6.3 remains at the core of the intertool PIF. Around that storage,

however, is a "storage manager" that provides a low-level interface to *access* that data. Surrounding the storage and storage access mechanisms is a *PIF toolkit* that provides a high-level, programmatic interface to an object-oriented view of profile data. TCAD applications can make direct use of the toolkit to (1) store or retrieve data, (2) exchange data with other tools, and (3) use toolkit data structures for new program development. Existing tools are integrated into the system via modification (to make toolkit calls), or by encapsulation via separate translation programs. Communication through a central database enables sharing of PIF-based utilities such as grid converters or PIF plotters. Furthermore, only a single writer or parser is needed to connect to ASCII PIF files, and through those files to different implementations of the PIF toolkit. The PIF toolkit plays a central role in the integration of new and existing TCAD tools, and the integration of these tools into a TCAD framework or system that interfaces with other design or manufacturing environments.

A key aspect of the intertool PIF is the interface provided by the toolkit (the bold line in Figure 6.4). As illustrated in Figure 6.5, the toolkit should provide for the connection of tools in several different programming languages. Furthermore, it is highly desirable that the interface be standardized to enable the implementation of compatible toolkits and to ensure that application tools that use the standard interface are themselves portable. For example, Section 6.4 of this chapter describes the PIF/Gestalt implementation of a PIF toolkit, where Gestalt itself insulates from underlying storage using the Ingres relational database. Similarly, the BPIF toolkit [63]. builds on the OCT data manager [64]. Both of these toolkits are examples where object-oriented databases are used for storage management. The PIF/Gestalt and BPIF toolkits do not yet implement a "standard" toolkit interface, but rather contribute toward the experience needed to define a standard such that compatible toolkit implementations based on commercial or other databases would be possible. A simple, although limited, implementation of a standard PIF toolkit interface might even use the ASCII PIF or other file formats for persistent storage, rather than a large-

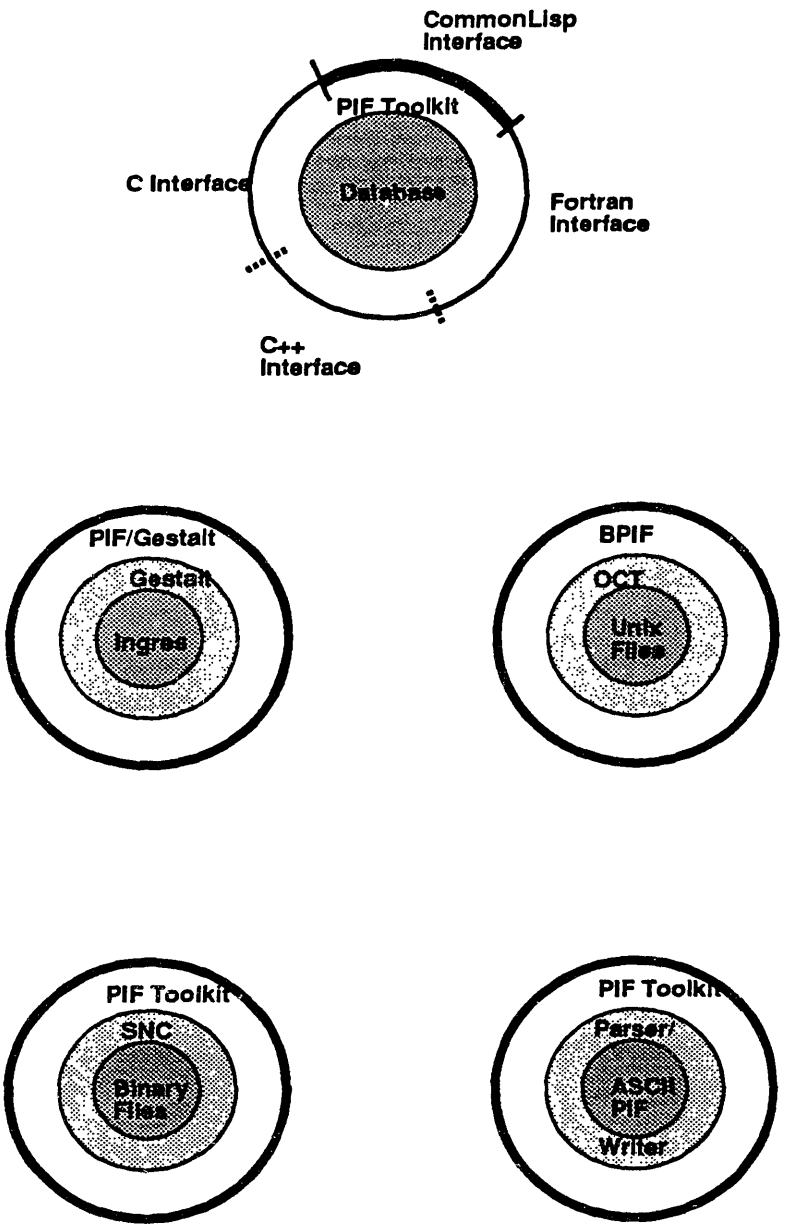


Figure 6.5: A "standard" programmatic interface with multiple programming language interfaces. A number of alternative implementations of the interface are possible, both on full-scale databases, and on ASCII or binary-formatted file storage.

scale object-oriented database. On the other end of the spectrum, the same interface might be implemented via communication between client programs and data servers such as that used in CHORD [65], or proposed in [66].

The intertool PIF can be viewed as bringing together into one place both TCAD data management and data structuring capabilities. The next section describes a conceptual model that guides the selection and definition of data objects used in the intertool PIF. The subsequent section then describes the definition of convenient functions for application programs to use to manipulate grids, geometries, and other objects appropriate to TCAD.

6.2 PIF Conceptual Model

We propose an object-oriented approach for the intertool PIF. The collection of objects used for the description of wafer and devices structures is termed the *PIF conceptual model*, and the definitions of these objects is the *PIF schema*. A goal of this work is to remain as close as possible to the intersite PIF defined in [53], so that both intertool and intersite versions of the PIF are compatible. The PIF schema, then, is directly based on the various geometry, snapshot, grid, and attribute constructs described in [53]. In order to define PIF objects and functions in a language- and database-independent way, an extended entity-relationship (E-R) method is adopted. Graphical E-R diagrams similar to those of Express-G [67] are used to depict PIF class definitions. The complete PIF schema is not presented here; fragments of the PIF schema (and the corresponding PIF toolkit) are used to illustrate our object oriented approach to wafer representation.

6.2.1 PIF Classes and Relationships

The term *object* (or *entity*) is used to refer to an encapsulation of data and behavior which describes some thing or event. An *object class* (or *entity class*, or simply

class) describes objects which share a common structure and behavior; intuitively, this corresponds to the abstract definition of some "kind" or "type" of object. A class is described in terms of named, typed information. The `Point` class in the PIF schema, for example, includes a "dimension," which is an enumerated data type with value `1`, `2`, or `3`, and coordinates "x," "y," and "z" of type `float`. The class definitions for the primitive geometric objects in the PIF conceptual model are depicted in Figure 6.6, where classes are shown as rectangular boxes, predefined classes (or built-in data types) such as `float` are shown as boxes with an extra line on their right, and enumerated data types are shown as dashed boxes.

In addition to encapsulating predefined data types, objects may also reference or be related to other classes. For example, the `Line` class is related to the `Point` class by the "points" relationship (denoted in Figure 6.6 by `A[min-size:max-size]`), which is an ordered collection, packaged either as a list or array, of two or more points. Relationships may also be single-valued, denoted via an arrow with no extra cardinality shown, or set-valued, denoted by `S[min-size:max-size]`. The *derived* relationship "line" from the `Point` class to the `Line` class (shown as a dashed arrow in Figure 6.6) describes all of the lines in which a particular point occurs. This relationship is the inverse of the "points" relationship.

6.2.2 PIF Object Graphs

The conceptual model or schema diagrams presented so far are depictions of PIF classes and the relationships amongst them. As such, they should be thought of as templates which can be instantiated, resulting in a graph of PIF objects. A crucial aspect of the intertool PIF is that such objects are not organized in a static file as in the intersite PIF, but rather are parts of a dynamic graph of objects. For example, the PIF *object graph* for a triangular face is pictured in Figure 6.7. Such a graph is an abbreviated depiction of particular objects and of the particular relationships between them. Application programs generate, manipulate, and traverse these instance graphs

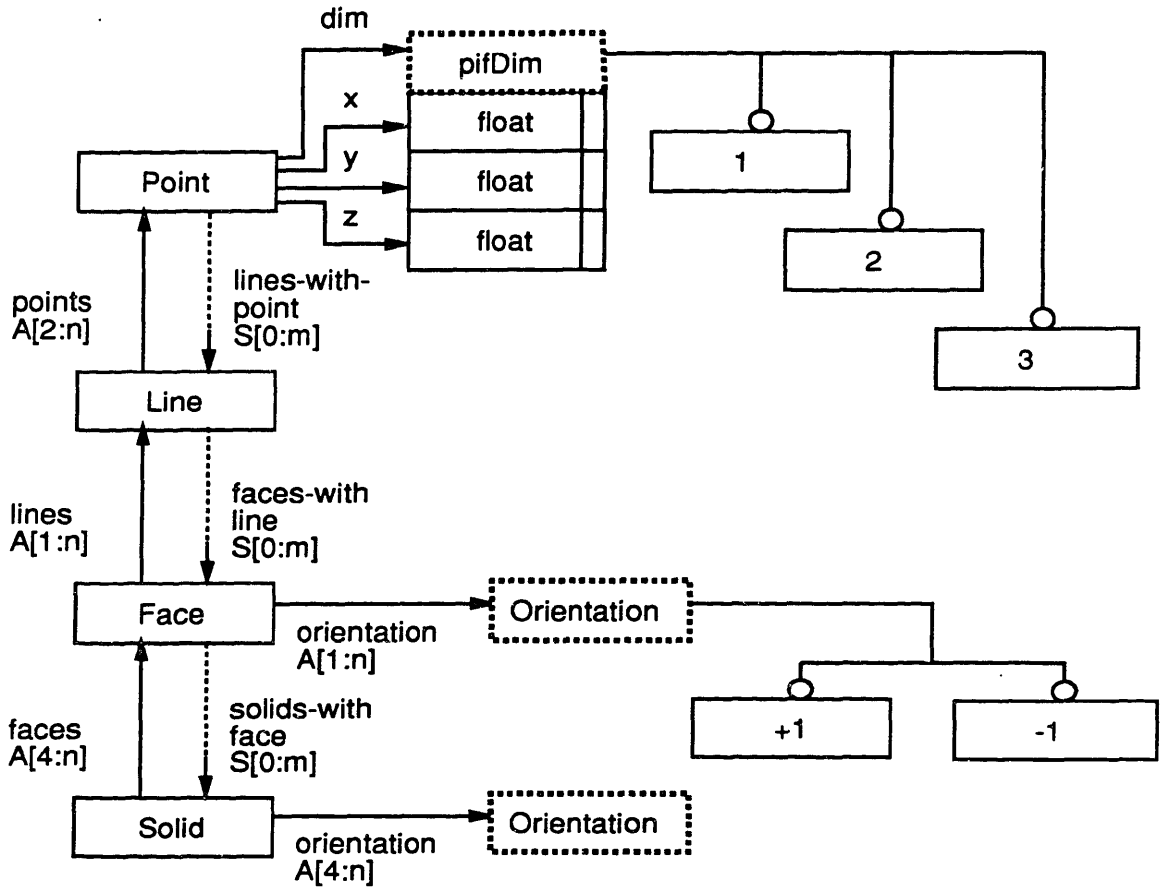


Figure 6.6: A PIF class definition diagram (based loosely on Express-G [67]) for the primitive geometries in the PIF conceptual model.

using toolkit functions.

6.2.3 PIF Class Hierarchy

Object classes in the PIF schema are organized in a superclass-subclass hierarchy. A subclass inherits all of the relationships of its superclasses. The PIF class hierarchy is illustrated in Figure 6.8. The `Point` class inherits from the `PrimGeo` class, and transitively from the `Geo` and `PifObject` classes. Because the `PifObject` class defines a string "name", any `Point` instance (and any other `PifObject` instance) may be named.

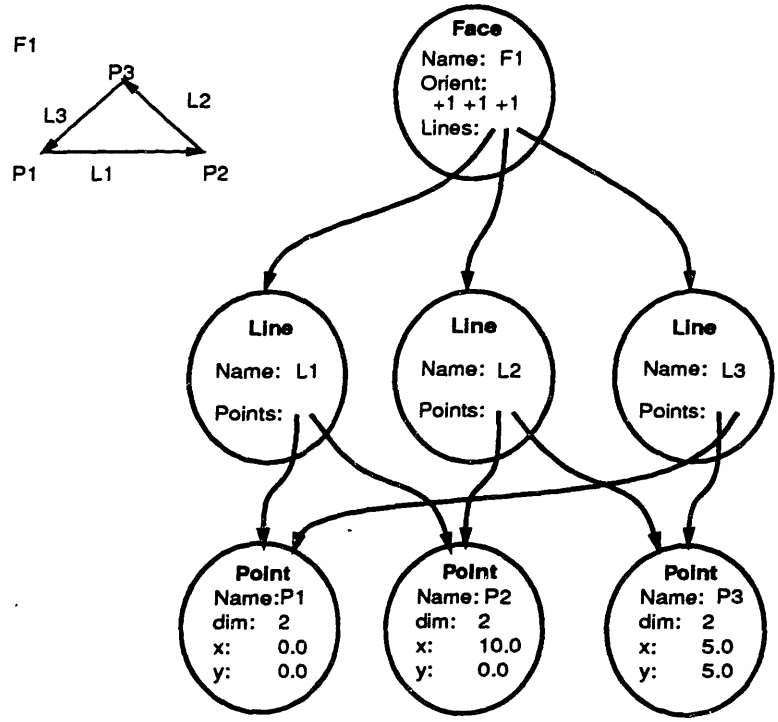


Figure 6.7: A graph of PIF object instances for a triangular face.

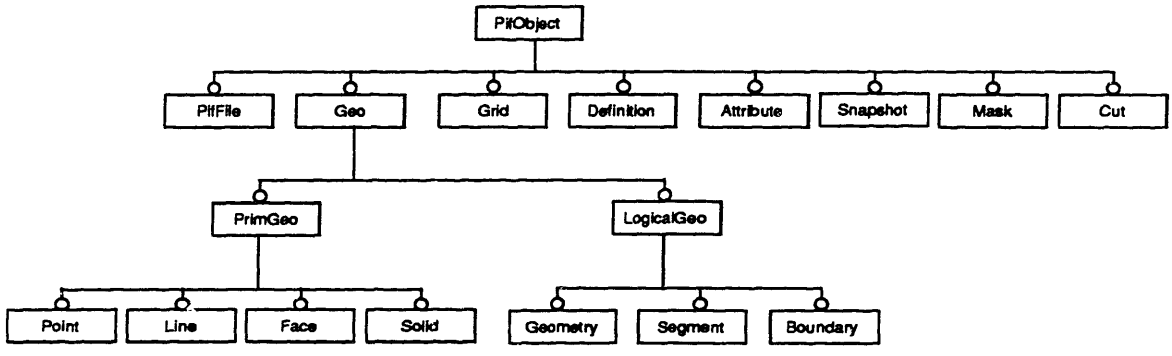


Figure 6.8: Basic class hierarchy in the PIF conceptual model.

6.2.4 PIF Attributes

Both the intersite and intertool PIF must support the definition of application-specific data or "attributes" and the association of attributes with device geometry. The classes defined in the PIF schema for attribute definition and association are shown in Figure 6.9. Each `PifObject` may be named by a string (which must be unique

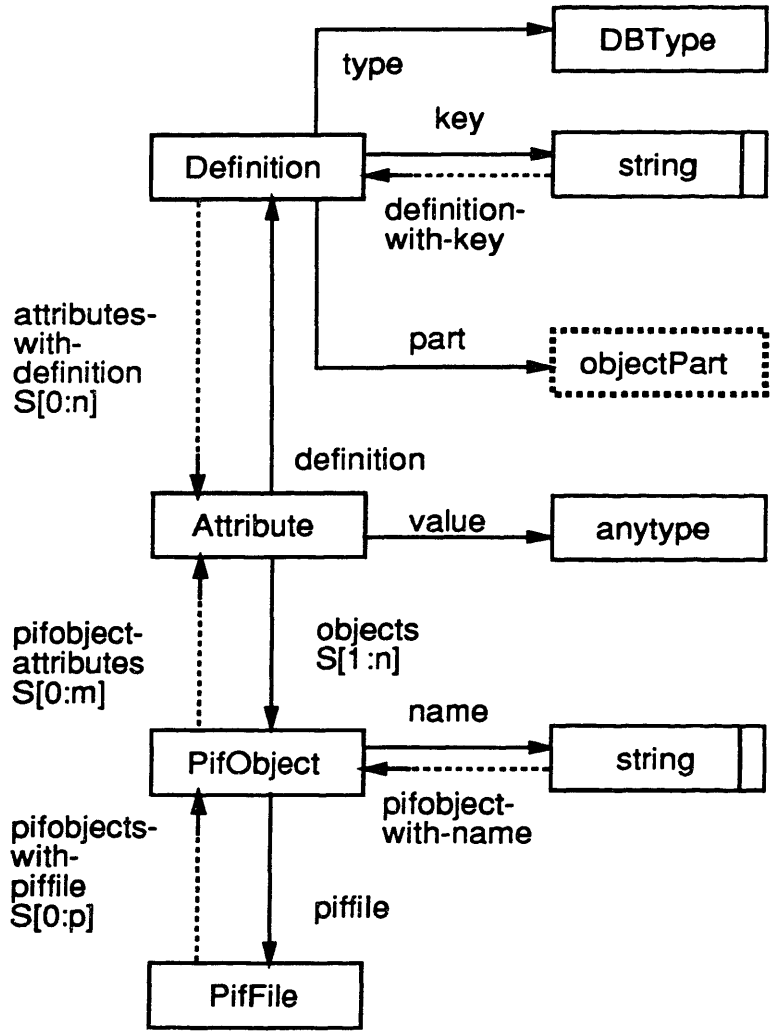


Figure 6.9: Definitions of PIF attribute classes.

within a `PifFile`). The `Attribute` class enables the association of a data `value` with

a set of `objects`; these objects can be geometries, attributes, or any other `PifObject`. Each `Attribute` must also refer to a `Definition` object. A `Definition` specifies the generic name or `key` for a group of similar attributes (such as "MaterialType"), and specifies the `type` of the data that each associated `Attribute` should contain. Many `Attribute` instances typically share the same `Definition`, and thus the same `key` (so that, for example, all "impurity-concentration" attributes in a `PifFile` can easily be retrieved). The `objectPart` of a `Definition` is used to establish a one-to-one correspondence between attribute values and some multi-valued part of the associated object. For example, an impurity concentration is usually defined on the "nodes" part of a `Grid` object.

6.3 The PIF Toolkit

The PIF conceptual model described in the last section can be mapped onto a set of functions for use in application programs. Requirements on the program interface provided by the resulting *PIF toolkit* are that it be similar across different programming languages, that it provide (nearly) seamless integration with the application programming language being used, and that is allow transparent access to many possible databases. The mapping to toolkit functions is guided by object-oriented programming ideas, particularly those embodied in the Common Lisp Object System (CLOS) [68]. Both Common Lisp and C versions of the toolkit functions are described; other language mappings (or "bindings") are possible, including Fortran and C++.

6.3.1 Toolkit Functions

Each class in the PIF schema has corresponding procedures for basic object manipulation (creation, deletion, and component selection and modification), associative access (*e.g.*, finding objects with a particular component value), and enumeration of

the objects of a particular class (the extent function). If a programmer knows the definition of a class, then the names of the functions for manipulating the objects directly follow and intuitive use of the toolkit is enabled. The rules guiding the choice of names for these routines are apparent from the examples summarized in Figure 6.10. Each relationship for a class will have a selector function. The selector function name

Function type	Common Lisp example	C example
Construct	`(setq p1 (make-point :name "MyPoint" :x 2.2))`	`p1 = make_point("MyPoint", ONE_D, 2.2);`
Delete	`(delete-pifobject p1)`	`delete_pifobject(p1);`
Select	`(point-x p1) -> 2.2`	`x = point_x(p1); /* 2.2 is assigned to x */`
	`(pifobject-name p1) -> "MyPoint"`	`char *name; name = pifobject_name(p1);`
Mutate	`(setf (pifobject-name p1) "Point1")`	`set_pifobject_name(p1, "Point1");`
	`(setf (point-x p1) 2.2)`	`set_point_x(p1, 2.2);`
Invert	`(lines-with-point p1) -> list of lines with point P1 in them`	`LIST s; s = lines_with_point(p1);`
Extent	`(listof 'point) -> list of all point objects in the PifFile`	`s = listof_point();`

Figure 6.10: Example functions provided in the PIF/Gestalt interface.

is explicitly defined in the schema for derived relationships, and for other relationships consists of the the name of the object class, a dash or underscore, followed by the relationship name. Mutator functions for all but derived relationships (which only have selectors) are via s `setf` in Common Lisp, or by a function beginning with "`set-`" in C. Deletion of objects is always through the `delete-pifobject` function. Each function returns a value (*e.g.*, the created object) so that function calls can be compounded together in a single expression. In the Common Lisp bindings, component values

needed for creation, selection, and mutation are specified with keywords (*e.g.*, `:name`). In the C bindings this is not possible, and all required arguments must be specified in a predefined order (though variable length argument lists allow omission of trailing optional arguments). In addition to these functions, the toolkit is augmented by specialized functions or methods that perform more complex or unusual functions. These include the `open_pif_file(filename, read_write_mode)` and `close_pif_file(pfile)` functions, for example. Some additional guidelines in the generation of a program interface from the PIF schema are considered below.

6.3.2 Built-In Data Types

One aspect of seamless integration with the application programming language is that the toolkit interface should take advantage of the built-in data types of the target programming language. In an earlier mapping to C toolkit functions, new classes were defined not only for the PIF object types (like `Points`), but also for basic types like `integers` (this was motivated by a desire to support detection of null values of all data types). This created a distinction between toolkit types and language types, and coercion between the two became necessary. As a result, typical toolkit functions became cumbersome to use (*e.g.*, function calls looked like `make_point(cptoSTRING("P1"), itoINTEGER(ONE_D), ftoFLOAT(2.38))`). Instead, we have found that it is more convenient to use the built-in types of the C language, or of whatever application programming language the toolkit is intended for.

6.3.3 Language-Specific Object Systems

While some languages, including C and Fortran, support only limited definitions of new types, several object-oriented programming languages include extensive mechanisms for defining and manipulating new types. In such cases, the PIF schema should be mapped as closely as possible to the object system of the target language. In the Common Lisp PIF toolkit, the Common Lisp Object System (CLOS) [68] provides

Function type	Generic Function example
Construct	`(make-instance 'p1 :name "MyPoint" :x 2.2)`
Delete	`(delete-instance p1)`
Select	`(slot-value p1 'x) -> 2.2`
Mutate	`(setf (slot-value p1 'name) "Point1")`

Figure 6.11: Generic functions provided in the Common Lisp/CLOS PIF Toolkit.

direct guidance in toolkit function generation (and can also be used in implementing the toolkit). In addition, CLOS suggests a number of *generic* toolkit functions in addition to the *specific* functions described earlier. Generic function examples for the PIF toolkit are shown in Figure 6.11. Another attractive language for a future PIF toolkit is C++, as it also supports a large degree of type extensibility.

6.3.4 Functional versus Structure Access

If the PIF toolkit is to enhance the portability of TCAD tools themselves, it is important that a "standard" interface be independent of any particular implementation (and of any underlying database). Such independence is greatly enhanced when creation, query, and modification of PIF object instances are restricted to function calls (or function call syntax). The functional syntax (*e.g.*, `point_x(p1)`) allows greater implementation flexibility than an explicit data structure access function (*e.g.*, `pifobject.pif.point.x`, as in the BPIF toolkit [63]). In one C-based toolkit implementation, the functional form might be a macro hiding underlying C structure definitions, while in another toolkit implementation the full functional form might be needed to execute database access code. The structure form exposes particular implementation decisions, and complicates alternative implementation (for example, the "." syntax cannot be overloaded in C++).

6.3.5 Toolkit Example: Face Construction

The example in Figure 6.12 shows the use of the Common Lisp PIF toolkit in the manipulation of geometric objects. The function `make-face-from-coordinates` takes an

```
;; MAKE-FACE-FROM-POINTS
;;  Assumes that the input points traverse the face
;; in a clockwise order, and that no points are
;; duplicated.
;;
(defun make-face-from-points (points)
  (let ((lines nil)
        (start-point (first points)))
    (dolist (point (append (rest points)
                           (list start-point)))
      (push (make-line
              :points (list start-point point))
            lines)
      (setf start-point point))
    (make-face :lines (reverse lines))))

;; Example use:

(setf new-face
      (make-face-from-points
       (list (make-point :x 0.0  :y 0.0)
             (make-point :x 10.0 :y 0.0)
             (make-point :x 5.0  :y 5.0))))
```

Figure 6.12: Example showing use of the Common Lisp PIF toolkit interface to generate the `Face` of Fig. 6.7.

ordered list of `Points`, and creates the `Lines` and `Face` corresponding to the instance graph shown in Figure 6.7. A C-language PIF toolkit version of the same function is shown in Figure 6.13. In both examples, the first `point` in the list is added again at the end of the list, and `lines` are created for each successive pair of points in the longer list. These lines are then passed to the `make-face` toolkit function and the created face is returned. The C example uses list manipulation functions provided by the C-language PIF toolkit, including `head` (returns the first item in a list), `dblist` (to create a list), `tail` (returns all but the first item in a list), `append` (to merge two lists),

```
/* make-face-from-points
 *
 * A C-language PIF toolkit example.
 *
 */
FACE make-face-from-points(points)
  LIST points;
{
  LIST lines = nullLIST();
  POINT start_point = head(points);
  LIST more_points = dbappend(tail(points),
                       dblist(start_point));

  for (p = head(more_points);
       !null(p=head(more_points));
       more_points = tail(more_points)) {
    lines = hitch( make_line( NULL, /* no name */
                              hitch(start_point,
                              dblist(p)) ));
    start_point = p;
    }
  return make_face( reverse( lines ));
  }

/* Example Use */
FACE new_face;
new_face = make_face_from_points(
  hitch( make_point( NULL, TWO_D, 0.0, 0.0),
  hitch( make_point( NULL, TWO_D, 10.0, 0.0),
  dblist(  make_point( NULL, TWO_D, 5.0, 5.0)))));
```

Figure 6.13: Example showing use of the C-language PIF toolkit interface to generate the `Face` of Fig. 6.7.

and `hitch` (to add an item to the head of a list). This example can be compared with the implementation of a similar function using the BPIF toolkit, presented in [63].

6.3.6 Toolkit Example: Attributes

In addition to the functions directly generated for creation of the objects and access of the relationships shown in the attribute class definitions of Figure 6.9, a number of "shorthand" attribute functions are also provided. These illustrate how basic toolkit functions relate to the conceptual model diagrams such as Figure 6.9, and how they

```
(make-definition :key "bbox"
                 :type "float array")

(defmethod set-bbox ((g geo))
  (let ((bbox (bbox-for-geo g)))
    (if bbox
        bbox
        (let ((new-bbox (calc-bbox g)))
           (make-attribute
              :definition
                (definition-with-key "bbox")
              :object (list g)
              :value new-bbox)
           new-bbox))))
```

Figure 6.14: Example showing the definition of the `bbox` attribute, and definition of a method that sets the bounding box for any geometric object.

can be combined to form more powerful functions. These extended functions include `(attributes-with-key key)` which is shorthand for `(attributes-with-definition (definition-with-key key))`, and `(attribute-key attribute)`, which is shorthand for `(definition-key (attribute-definition attribute))`.

A second example demonstrates two aspects of the PIF toolkit. First, the use of attribute objects and relationships defined in the conceptual model (Figure 6.9) are illustrated. This example also illustrates the use of polymorphic utility toolkit functions or *methods*. In Figure 6.14, a `Definition` for bounding box (or "bbox") attributes is first created. Next, the `bbox-for-geo` generic function looks to see if there is already a bounding box attribute attached to the object, and if so, returns it. If not, a new bounding box is calculated, attached to the object for future reference, and returned. In this example, the `calc-bbox` function is a method which knows how to calculate the bounding box depending on the kind of geometric object that is passed in as an argument.

6.4 PIF/Gestalt Implementation

6.4.1 Object-Oriented Database

PIF/Gestalt is a test implementation of the PIF toolkit. The functions described above are implemented on top of the Gestalt object oriented database [69] as illustrated in Figure 6.15. Gestalt has been developed to support MIT's Computer Aided

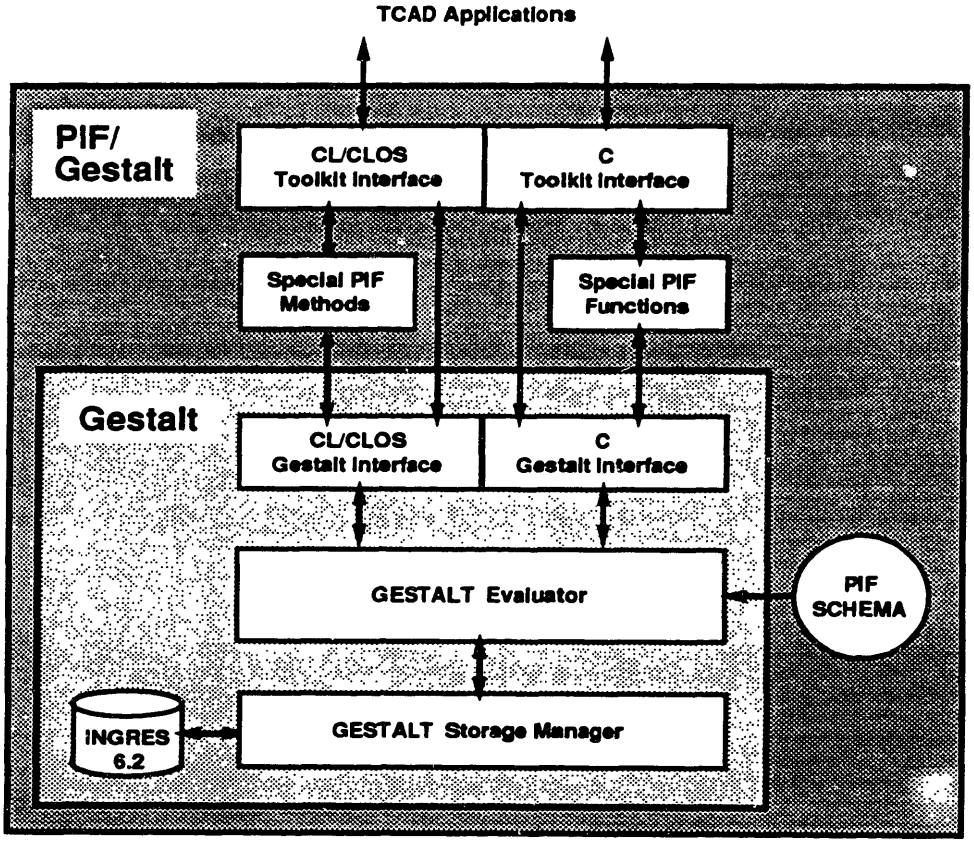


Figure 6.15: Implementation of the PIF toolkit on the Gestalt database.

Fabrication Environment (CAFE), a comprehensive computer integrated manufacturing (CIM) and CAD system devoted to semiconductor design and fabrication [6], of which a TCAD environment is an integral part. Additional advantages of Gestalt are that it (1) inherently provides an object-oriented programming interface to persistent

```
(make-inherited-dbtype
 :name "POINT"
 :overview "A point in space."
 :dbslots (make-dbslots-list
  '((:name "dimension"
     :dbtype ,(dbtype-with-name "INTEGER"))
    (:name "x" :dbtype ,(dbtype-with-name "FLOAT"))
    (:name "y" :dbtype ,(dbtype-with-name "FLOAT"))
    (:name "z" :dbtype ,(dbtype-with-name "FLOAT"))))
 :superclasses (this-and-its-superclasses "PRIMGEO"))
```

Figure 6.16: Generation of the schema entry for the PIF Point object.

storage; (2) insulates the application from the specifics of the storage system and enables the use or substitution of different storage systems; (3) is currently based on Ingres (from Ingres Corp.), providing for robust database capabilities, such as locking, data recovery, etc.; (4) provides support for multiple languages, including C and Common Lisp; and (5) provides mechanisms for the specification and modification of application schemas. This last point enables the direct, automatic, and consistent generation of the PIF toolkit from the PIF schema, and merits further attention.

6.4.2 Gestalt Object Definition

To implement the PIF toolkit, the formal conceptual model of the PIF is expressed as a short program that makes Gestalt object class definition calls. Object types or classes are defined by creation in Gestalt of DBTYPE objects. The information about an object to be managed by Gestalt is specified in a list of DBSLOTS for the object type, where a "slot" is a named relationship or collection of attachments to other objects, or a value contained within an object. As an example, the call to Gestalt to generate the schema entry for the Point type is illustrated in Figure 6.16. Gestalt supports the supertype-subtype hierarchy among defined types, and all of the slots defined for a supertype are inherited by its subtypes. Enumerated data types are not directly supported by Gestalt. Derived relationships that are inversions of slots are

supported by Gestalt (*e.g.*, the `lines-with-point` function), while functions for other kinds of derived relationships must be generated by hand (*e.g.*, `attributes-with-key`).

6.4.3 Toolkit Function Implementation

Once the schema defining the set of PIF objects has been created as in Figure 6.16, most PIF toolkit functions are generated automatically by Gestalt (all but 17 of nearly 150 toolkit functions). Most of the toolkit functions are generated as small code fragments or macros that make calls at runtime to an internal *Gestalt evaluator*. This evaluator accepts application requests and generates the calls required by the underlying storage system to execute the request. Calls to the evaluator are of the form `gestalt_eval(operation_tag, type_tag, arguments...)`. Generic functions such as `make_instance` can be implemented as macros that specify the `operation_tag` (*i.e.*, `CREATE`, `DELETE`, `MODIFY`, `SELECT`, and `LISTOF`). More specific PIF functions such as `make_point` are macros that hide both operation and type tags from the programmer. The use of an internal evaluator eases the implementation of the toolkit, and further helps to ensure that PIF objects are handled uniformly within the toolkit.

6.4.4 Database Transparency

An important distinction between the BPIF and PIF/Gestalt implementations is the relative transparency of the underlying database. In BPIF, there are both in-memory and in-OCT copies of objects, and the program must request that an object be stored into or retrieved from the database. In PIF/Gestalt, on the other hand, the storage or retrieval of information from the database need not be explicitly requested. While PIF/Gestalt internally uses an in-memory, cached copy of a database object when an object is accessed by a program, the program sees only a single handle to the object, and movement of information between the cache and the storage system is handled automatically by the PIF/Gestalt creation, deletion, and slot access and modification functions. A limitation in the current implementation is that all PIF objects must be

persistent (that is, stored in the database), with the storage and efficiency overhead this implies. A good compromise between these two approaches would be to have only one style of interaction with a PIF object, but allow the object to be specified as transient or persistent at declaration or creation time.

6.5 PIF/Gestalt Application Examples

To test the object model, the toolkit functions, and the PIF/Gestalt implementation, several prototype applications have been implemented as summarized in Figure 6.17. These interfaces and applications form a subset of a full-scale technology CAD system (such as that pictured in Figure 6.4) that is sufficient to explore and demonstrate the kinds of use expected of the PIF toolkit.

6.5.1 PIF Parser

The intersite PIF definition [53] is flexible and powerful, and is well-suited to the *output* (or writing to a file) of wafer and device structure information. The *input* (or parsing from a file) of the PIF format, on the other hand, is complicated by this flexibility, and few full-functioned parsers of the format have been demonstrated. Using the PIF/Gestalt toolkit, a parser that handles the full ASCII PIF format, with the exception of the *reference* construct, has been written. The PIF parser was implemented as a recursive-descent parser in Common Lisp. It makes use of the LISP reader to access the bulk PIF information, and then makes PIF toolkit calls to generate the object instance graph, as well as to retrieve named PIF objects during graph generation. By making use of the PIF/Gestalt toolkit, the parser was written in 750 lines of code.

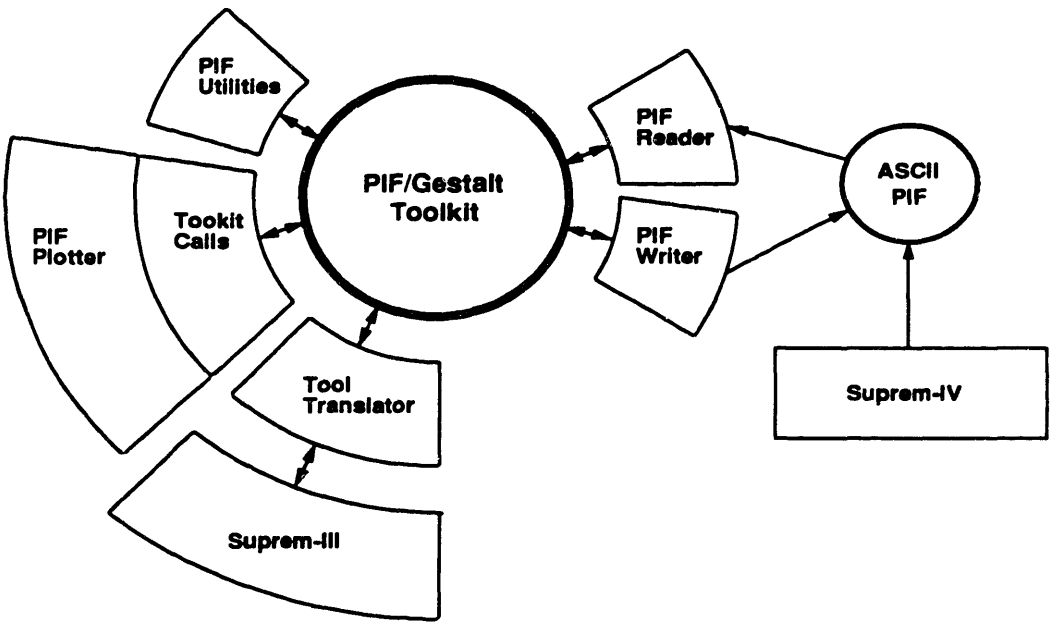


Figure 6.17: Prototype tools based on PIF/Gestalt. Arrows indicate the flow of wafer or device structure information.

6.5.2 Use with Existing Tools

A C-language version of the PIF/Gestalt toolkit was used to construct two small PIF conversion programs for Suprem-III. The *sup2pif* program converts between the internal save structure of Suprem-III [24] and the PIF/Gestalt database, and required 441 lines of C code. The *pif2sup* program extracts information from a database `pifFile` to write a save file for use by Suprem-III, and required 272 lines of C code.

A connection with the Suprem-IV process simulator [70] was established to test the 2-D PIF toolkit objects. A modified version of Suprem-IV was used to directly

generate ASCII PIF output.† The boron implant diffusion example "boron.in" provided in the Stanford Suprem-IV distribution produces a final output structure file from Suprem-IV that is 13340 bytes long; the corresponding PIF file generated by the modified Suprem-IV program requires 13295 bytes. After parsing into the PIF database, approximately 39 kbytes (in various Ingres tables) are required by Gestalt to store the same information. The cpu time required to parse the PIF file is 119 seconds. Much of this time (89 cpu seconds, or ≈75%) is due to object name lookups required by the ASCII PIF.

6.5.3 Use in New Tools

A PIF Plotter program was written which uses information stored in the PIF database to generate graphical displays of PIF objects. The plotter draws (and labels) 1-D and 2-D geometric objects, and can plot attributes defined on grids on those geometries using X-Y and contour plots. The plotter requires 272 lines of Common Lisp code, and generates input to the Giraphe3 plotting program [71, 72]. The plotter also makes use of PIF utility routines totalling 436 lines of code (*e.g.*, for calculation of bounding boxes).

6.5.4 Efficiency Considerations

A simple example illustrates limitations in the prototype implementation of PIF/Gestalt. A small program was used to generate and access 10,000 point objects (each with x, y, and z coordinate information). The storage required by Gestalt is 854 kbytes, the creation time is 1066 seconds cpu time, and the retrieval time (for first access) 171 seconds on a Sun-3/260 with 16 Mbyte memory. The primary storage system used within Gestalt is a relational database. While the PIF/Gestalt implementation compares favorably with other such layered approaches [73], creation and access time is

†Implemented by Goodwin Chin, Stanford University.

currently too long for direct use by most TCAD applications. Implementation of the PIF conceptual model and toolkit functions proposed in this chapter using emerging commercial object-oriented database technology is an interesting area for future investigation, and promises to yield acceptable performance for widespread use by TCAD tools and frameworks.

6.6 Conceptual Model Experiments

As mentioned in Section 6.2, an important goal was to keep the set of objects provided by the intertool PIF compatible with those defined in the ASCII PIF. In some cases, this meant accepting the names used for object types, where more appropriate names might be preferable. For example, the PIF uses the terms "point" and "line" to denote what are really topological objects better called "vertex" and "edge", respectively. More importantly, the ASCII PIF defines constructs and an organization that are not necessarily appropriate for direct program use. Investigation of two such cases follows.

6.6.1 ASCII PIF Lists

In the ASCII PIF [53], it is possible to bundle together some collection of objects as a single named "list". To investigate the implications of this capability in the intertool PIF, a `PifList` object which contains any number of `PifObject` members was defined. The `PifList` was defined to inherit from the `PifObject` class, so that it was a full-fledged PIF object which could be named and referred to on its own, and which could have attributes associated with it (as in the ASCII PIF). With this object, it was possible to build an instance graph that closely mimics the intersite PIF. The graph shown in Figure 6.18 corresponds to the intersite PIF fragment in Figure 6.19.

In using a PIF toolkit that includes the `PifList` object, undesirable ambiguity is

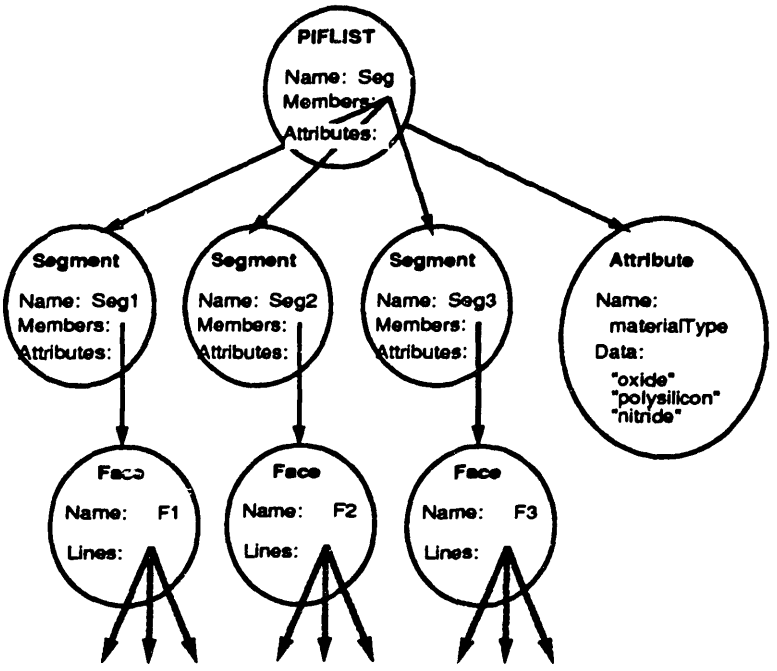


Figure 6.18: Network created by parsing the PIF of Figure 6.19 into a PifList.

```
(segmentlist
  (segmentlistname "Seg")
  (facenamelist F1 F2 F3)
  (materialType "oxide"
    "polysilicon" "nitride"))
```

Figure 6.19: ASCII PIF text showing association of attributes with a list of geometric objects.

introduced as to the location of attributes for a given object. Normally, one expects to be able to follow a single `attributes` relationship to get all of the attributes associated with some geometric object. However, the object may also be a member in one or more `PifLists` having an attribute with values that are in one-to-one correspondence with the members of the `PifList`. In the example shown, if a user wants to get the `materialtype` for `Seg1`, he or she would have to look in more than one place. Furthermore, consistency becomes difficult to maintain because of the implicit correspondence of attribute data with objects: deleting the second `Segment` might shift the association of the "polysilicon" attribute to an incorrect segment. Based on this experience, the use of `PifLists` for grouping geometric objects is not recommended; the semantics of the intersite PIF should be changed so that the PIF list notation be used only as shorthand for specifying multiple attribute associations. In the example above, three separate instances of the `materialType` attribute should be created, each containing only one piece of data, and each associated with an individual segment.

6.6.2 ASCII PIF Hierarchy

The intersite PIF is a static file-based format. In order to (1) give TCAD programs "random access" to the PIF, and (2) decrease the storage required by a verbose character-oriented format, an alternative that was considered early in the development of an intertool PIF was to implement what might be described as an "in-memory" mapping of the ASCII PIF. Here, the structure of the ASCII PIF is maintained exactly: what was an enclosing parenthesized form in the ASCII PIF is transformed directly into a parent object containing as children what were the enclosed forms. This approach is particularly tempting for a LISP interface, where one can use the LISP reader directly to build an in-memory list or tree representation of an ASCII PIF file.

In order to model such a parent-child tree of objects in a PIF toolkit, a `PifNode` class is added that is a subclass of the `PifObject` class. Each `PifNode` has exactly

one parent, and each `PifObject` may have any number of children. During a parse of an ASCII PIF file, the parent-child links are established to mimic the textual organization of the ASCII PIF. In using these parent-child relationships one finds that locating and manipulating objects based solely on parent-child links is difficult and inconvenient. As with the `PifList` object, there is a great deal of ambiguity in the location of information within the parent-child tree, so that retrieval of attributes or geometry information may require traversal of large portions of the tree (though scanning of bulk data can be avoided). The conceptual modeling and use of specific relationships between geometric and other PIF objects, as described in Section 6.2, is much preferable.

6.7 Conclusions

The object-oriented approach to semiconductor wafer and device structure representation has proven to be successful in defining a uniform, consistent, and easy-to-use intertool PIF toolkit. A formal conceptual model of the objects in the intertool profile interchange format (PIF) has been presented which defines a number of PIF "objects" and the relevant "relationships" between PIF objects. The object model can then be mapped onto multiple programming languages to provide TCAD tools with a high-level, programmatic toolkit interface to create, access, and manipulate semiconductor profile information. A test implementation of the toolkit, PIF/Gestalt, demonstrates the C and Common Lisp functions in the PIF toolkit, and eases the programming difficulty in interfacing to and generation of TCAD tools and utilities. A "standard" toolkit interface such as that proposed here will facilitate the integration of TCAD tools together into large-scale systems and frameworks [74, 75].

At the same time, there are limits to the flexibility and power provided by the PIF toolkit. The PIF conceptual model and resulting programmatic interface provide data-oriented access to wafer geometry and attribute information via a functional in-

terface. "High-level" functionality, where potentially large amounts of computation and object manipulation occur, is not provided by the toolkit. A great deal of current research, development, and standardization work to define geometry, field, and attribute "servers" is focusing on these issues [66].

Part III

Tools for Process Design

Chapter 7

PFR-Based Simulation Manager

In any design system, a crucial capability is the simulation of the behavior of a design to explore the design space or test hypotheses. When the designed artifact is a fabrication process, the prediction of the cumulative effect of a process on a wafer during processing is essential. The mechanism by which designers specify the fabrication process within CAFE (Chapter 3) is the process flow representation (PFR), described in detail in Chapter 5. It is important that designers be able to simulate based on a single, *unified* representation of the wafer. This chapter describes a *Simulation Manager* which enables a designer working with the PFR to perform process simulation to evolve, verify, or understand process designs.

The architecture of the Simulation Manager is described in Section 7.1 first, from a user perspective, and second, with an implementation view. The mask and cross section model supported by the Simulation Manager is presented in Section 7.2. Section 7.3 describes the process translator component of the Manager. Data and task management requirements, including simulation sharing and minimization, representation and handling of photolithography, and management of resimulation are discussed in Section 7.4. Finally, conclusions based on the implementation of the Simulation Manager are offered in Section 7.5.

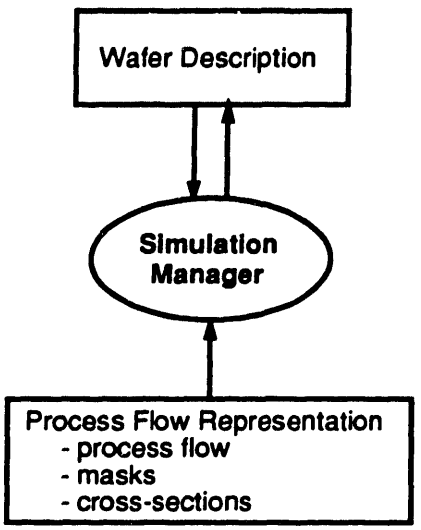


Figure 7.1: Structure of the Simulation Manager.

7.1 Architecture

The fundamental role of the Simulation Manager is to transform, under user control, descriptions of a semiconductor wafer or device structure, given a fabrication process description (as shown in Figure 7.1). In this sense, the Manager itself is conceptually a highly interactive, PFR-based process simulator.

The Simulation Manager performs two key tasks to aid in the simulation of a fabrication process, as illustrated in Figure 7.2. The first of these is *process translation*. Given a generalized process flow description, mask definitions, and cross section definitions, the task is to produce input for some process simulator. Generation of input files to process simulators, however, is not sufficient for the effective simulation of a process. The second critical task is to then *manage* the execution of process simulations. Simulation management requires the handling of multiple cross sections (especially when dealing with one-dimensional process simulators), performance of incremental simulation or resimulation of the process so as to help pinpoint difficulties in the simulation, interactive control of the simulation by the designer, and finally, minimization of the total time spent in simulation. Each of these is addressed by the

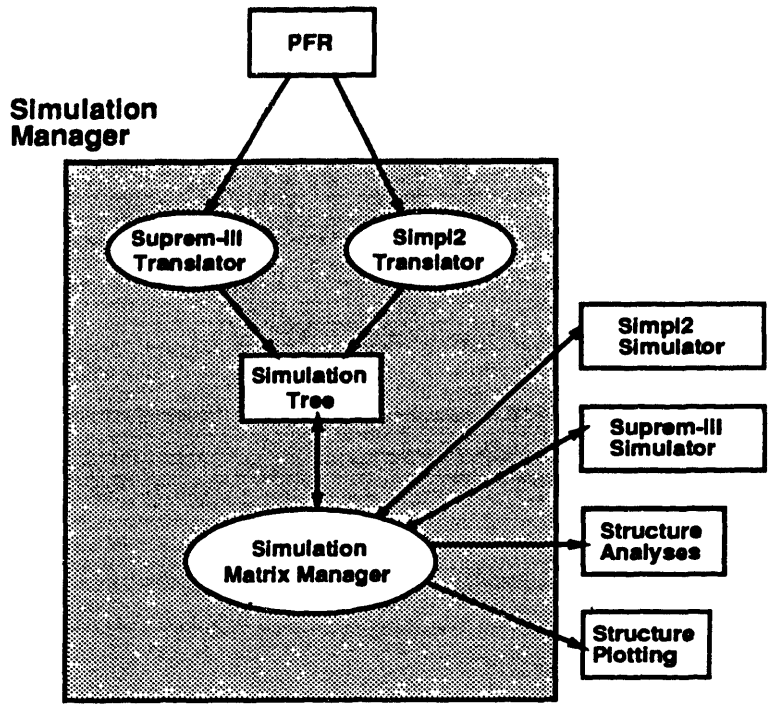


Figure 7.2: Architecture of the Simulation Manager.

Simulation Manager.

The Simulation Manager has been developed to provide general support for simulation of semiconductor processes written in the PFR. This implementation of the Simulation Manager supports both one- and two-dimensional process simulation capability. In this chapter, the focus is on the general issues and specific concerns of one-dimensional simulation (primarily at the treatment level) using the Suprem-III process simulator [24]. Two-dimensional simulation (primarily at the change in wafer state level) is accomplished via SIMPL-2, and is discussed in Chapter 8.

The Simulation Manager is implemented in Kyoto Common Lisp (KCL) [76], a publicly available, portable implementation (via compilation to C) of the Common Lisp definition [77]. The Manager makes heavy use of the PCL implementation of the Common Lisp Object System [68]. In addition to these system level components of CAFE, the Simulation Manager also uses the Fabform character-cell, template-based

interface toolkit provided by CAFE [12]. While many aspects of the user interface are restricted by the lack of a graphical user interface, the capabilities of Fabform have been adequate. Also, the textual terminal interface has proven to be easily portable, and can operate under conditions where graphical interfaces are less readily available (for instance, to guide or evaluate simulations from a text terminal connected over a modem). The Manager does not currently use or depend upon the Gestalt database component of CAFE. The implementation (excluding the above components) totals approximately 6000 lines of code, and consists of the following modules shown in Table 7.1.

module	function	lines of code
sim-node.lisp	generic simulation tree manipulation	1269
sup-node.lisp	Suprem-III specialization to sim-nodes	75
simpl-node.lisp	SIMPL2 specialization to sim-nodes	87
fl.lisp	Core PFR evaluator †	836
fl-cws.lisp	Adjunct CWS evaluator	113
fl-treatment.lisp	Adjunct Treatment evaluator	83
inexact.lisp	Inexact value package	500
suprem-trans.lisp	Suprem-III translator	465
simpl-trans.lisp	SIMPL2 translator	782
diffusion.lisp	Diffusion length estimator	178
sim-manage.lisp	Simulation Manager Interface	1510

Table 7.1: Code modules in the Simulation Manager.

7.2 Mask and Cross Section Model

In addition to descriptions of processing steps (the treatments and changes in wafer state), also necessary for the simulation of the process are definitions of masks and cross sections. The requirements on mask and cross section representation are (1) they should correspond well with the way that process designers think about masks and cross-sections, (2) they should be independent of particular simulators (that is, they should be based on what cross sections and masks fundamentally are), and (3) they should be computer-manipulable in order that the Simulation Manager can make use of them.

7.2.1 Mask Model

A *mask* is generated from some geometric combination of layout layers. A typical physical mask consists of a glass plate with clear and dark areas on it to block or transmit light during photolithography. Often, the dark areas are in one-to-one correspondence with some named layer in a circuit layout, and most of the mask is clear (a *clear field* mask). In a *dark field* mask, the mask is clear where a layout is colored and dark everywhere else.

The generation of actual masks are made through a non-trivial sequence of computational and physical process steps similar to those used in wafer processing. The fundamental intent is to transfer *layout layer* information to the physical mask. A mask having a pattern that is in one-to-one correspondence with a layout is the usual goal. The patterns may, however, be the result of geometric manipulations of one or more layout layers, both intentional (the merging of device layers with blocking edges, optical patterns, alignment marks, misalignment verniers, *etc.*) and unintentional (the result of optical artifacts including shrinking and bloating). For this work, we restrict the definition of the pattern on a mask to be the geometric merge or union of one or more layout layers. A dark field mask is represented as an inver-

sion of the merged layout. The mask definition, then, is restricted to the geometric union of one or more layout layers, with a possible inversion at the end. That is, $M = L_i + L_j + ... + L_n$ or $M = \overline{L_i + L_j + ... + L_n}$, where + indicates geometric union, M is the mask, and $L_i \cdots L_n$ are layout layers.

7.2.2 Cross Section Model

A definition of a *cross section* that corresponds to the mental picture used by the process designer is also required. When a designer speaks of the "gate section", for example, he or she usually means the area (or any area) lying under both the diffusion and poly layout layers. That is, a one-dimensional cross section is defined by some intersection of layout layers. The representation of a one-dimensional cross section, then, is an exhaustive listing of the layout layers that are present (or *dark*) at that point in the layout. That is, $S = L_i * L_j * ... * L_m$, where * indicates geometric intersection, S is the section, and $L_i \cdots L_m$ are layout layers. For convenience, if a layout layer is not mentioned, it is assumed that the cross section must *not* contain that layer. For instance, if L_x is a layer present elsewhere on the layout (and not in section S), then S could also be defined and understood to mean $S = L_1 * L_2 * \overline{L_x}$.

7.2.3 Use of Mask and Cross Section Definitions

The representations described above have a number of advantages. First, it is possible to define a mask or a cross section in terms of *generic* layouts. That is to say, no actual layout need be specified or developed in order to produce meaningful simulations, nor must the simulation be tied to any particular circuit layout. Instead, the descriptions work well for *mask sets* which follow the same layer naming, and to any layouts with generic sets of layer crossing (or devices) on them. The descriptions are also comparatively natural, and make it possible for the designer to guide and direct simulations with statements like "simulate the NMOS gate section up to the channel implant step".

For the purposes of process translation, the above mask and section representations have the further advantage of easily supporting selective translation per cross section. During processing, the actual point of differentiation on the wafer comes when light (or some other energy or mass beam) strikes some regions of the wafer and not others, depending on whether or not the beam is blocked by the dark region of the mask (either physically, or logically as in direct beam writing). The problem, then, is to determine during translation whether or not the particular cross section is *exposed*. The above representation makes this determination particularly simple. The section is exposed if and only if there is a geometric intersection between the inverse of the mask and the cross section definitions. That is, if any layer mentioned in the definition of a dark field mask is also listed in the layers that are present for a cross section, then the section is exposed, while if any layer mentioned in the definition of a clear field mask is also listed in the layers for the cross section, then the section is *not* exposed.

7.2.4 Higher Dimensions

The mask definition used here is independent of the dimensionality of the simulation to be performed. The *point* specification of layout intersection in the cross section definition above, on the other hand, is suitable only for one-dimensional (drill-hole) simulation. It is possible, however, to extend cross sections to the definition of cut-lines (generating two-dimensional simulation regions) and entire layout areas (generating three-dimensional simulation regions). In essence, a cross section definition is nothing more than a subset of a full layout, expressed in a position-independent fashion. The one-dimensional section definition above corresponds to a zero-dimension abstraction or condensation of a layout. A two-dimensional cross section definition corresponds to a one-dimensional abstraction or a layout, and would consist of *ranges* of the presence of a layout layer (exactly like that extracted from a layout by a cut-line in SIMPL-2). And a three dimensional cross section corresponds to some area of a

full layout. The modeling of exposure where edge effects are important and physical simulation is necessary is much more difficult [27].

7.3 Process Translation

Process *translation* is similar to the process "interpretation" or "compilation" from one form into another implemented by Kager in PI/C [78]. In that case, however, the process description was intended solely for simulation, and the target description consisted of calls (either directly or via construction of another program) to the Fabrics simulator [79]. This work differs substantially in that the PFR is a general purpose description of the process, and several different tools require information drawn from the PFR. This approach is similar to that taken in SCHEMA for electronic design with its emphasis on uniform design representations [80]. There are several subtasks and issues involved in such process translation, including mechanisms for access to PFR information, conversion of that information at the change in wafer state or treatment level into statements that the target process simulator understands, and generation of data structures suitable for further use and manipulation by the Simulation Manager. For one-dimensional simulation, additional complexity is required to account for the effect of masking on a cross section.

7.3.1 Access to PFR Information

The basic interface to PFR information is through the core PFR and adjunct evaluators (*i.e.*, the CWS and treatment evaluators). This interface might be used in two ways: incrementally or monolithically. In the monolithic approach, recursive application of the core evaluator is used to generate a complete operation instance tree. In the incremental case, an application interpreter calls the evaluator only when a particular PFR fragment is to be examined. The Suprem-III translator could be constructed using either approach.

In the monolithic case, translation would be based on the instantiated operation tree. An advantage of this approach is that PFR evaluation could take place separately from translation. Loading a textual PFR into a persistent database, for example, would allow the operation tree to be shared between multiple applications (the fabrication and simulation system, in particular). A disadvantage of this approach is that many of the textual aspects of the process description are lost to the translator (*e.g.*, the names of definitions, the existence of function applications, and the arguments of those calls), because these language-like forms are generally removed by the core evaluator. A compromise used by McIlrath in the fabrication interpreter [15] is to create database operation objects that include the textual PFR fragments that generated them. Another important disadvantage of this approach is that generation of large numbers of persistent objects is slow in Gestalt, and would significantly slow the process translation task.

The alternative chosen in the Suprem-III translator is incremental PFR evaluation during translation. An operation to be translated (beginning with the top-level flow) is partially evaluated until an operation object can be returned. If the operation contains a non-empty treatment description, the treatment adjunct evaluator is called, and Suprem-III code is emitted corresponding to the returned treatment objects (a code conversion phase). If the operation does not contain a treatment but does contain a non-empty change in wafer state attribute, the CWS adjunct evaluator is called, and corresponding code is constructed. If both attributes are empty, each sub-operation in the body of operation is then translated; if the body is empty, no code is generated.

The incremental translation enables access to textual PFR information, and enables the structure of the interpretation, and not just the operation structure, to guide the simulation tree generation. In particular, the simulation tree has new nodes inserted not only when new operations are encountered, but also (optionally) whenever a function is invoked as part of an operation body. Furthermore, the names used

in the textual PFR are retained and used both in the Simulation Manager interface and in comments in the generated Suprem-III code. Such name correspondence is especially important for understanding and debugging a process.

7.3.2 Code Conversion

The heart of the translation task is conversion of information found within a treatment or change in wafer state object into textual statements appropriate for the target simulator. Generally such conversion is straightforward: the object is queried for information, necessary unit conversions are performed, and output statements adhering to target simulator syntax are formatted and output. In some cases, the emission of code is sensitive to the state of the wafer during the operation. For example, if resist is known to be at the surface of the wafer for the cross section, ion implantation is translated as a comment rather than as an Suprem-III implant statement.

The knowledge of how PFR converts to Suprem-III is thus expressed by the Common Lisp code implementing the translator. In cases where the predefined translation is not suitable (*i.e.*, when unusual or special case Suprem-III statements or parameters are needed to better model a particular step), the `:suprem3-code` stranger attribute of an operation in the PFR can be used to directly specify and override the translation. The computational mechanisms provided by the PFR, then, enable a limited amount of end-user programmability in the generation of simulator code. However, extensive modification or addition of translation capability requires the modification of translator source code. An alternative approach demonstrated by Wenstrand [30] is to require that the user directly define all simulator translations within operation or process objects. Because process objects in PDA may inherit from generalized, predefined objects, the two mechanisms result in nearly identical capability. PDA has the advantage of uniformity of simulator code translation, while the Simulation Manager decouples the writing of PFR descriptions (in which programming features are intentionally restricted) and simulation translation code (in which the full power

of CommonLisp is available).

7.3.3 Simulation Tree Data Structure

The essential data structure constructed and manipulated by the Simulation Manager is a *simulation tree*. The result of process translation for some specific cross section is the simulation tree, which then contains the information needed to drive the target process simulator.

The structure of the simulation tree is shown in Figure 7.3. The basic object in

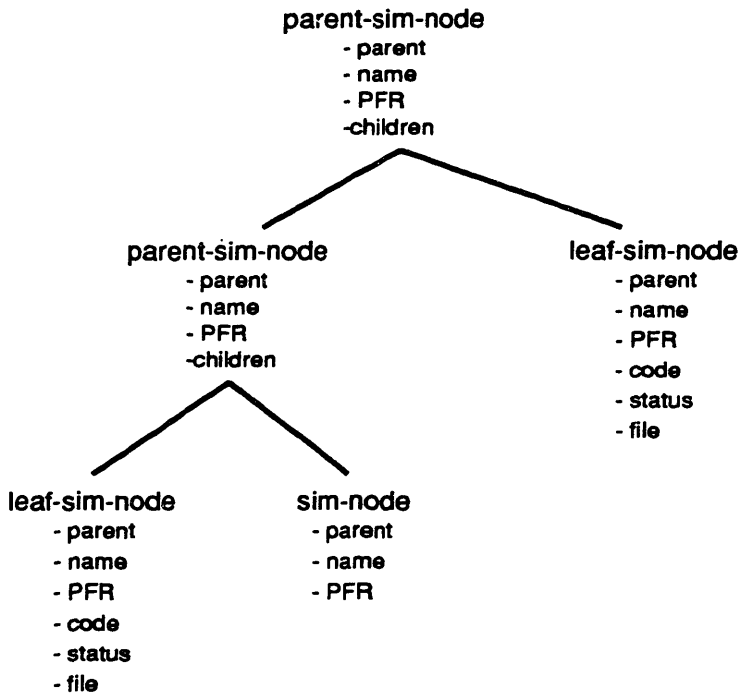


Figure 7.3: Example structure of a simulation tree.

the tree is a `sim-node` corresponding to some operation (or function invocation) in the PFR. Each `sim-node` contains the PFR code fragment that generated the subtree rooted at the node, a name derived from that code fragment (the name of an invoked definition or function), and a pointer to its parent node in the tree. A `parent-sim-node`

has all of the slots of the `sim-node`, and is distinguished in that may also have an ordered list of `children`. The `leaf-sim-node` is another specialized `sim-node`, and is the only node type that can contain code fragments or simulation results. The `code` slot in a `leaf-sim-node` is a list of Suprem-III statements (strings); the `status` indicates the need to simulate (or resimulate) the particular node; and the `file` slot points to a file containing simulation results. In this implementation, a simulation result is stored as a Unix file in the native format of the target process simulator. Alternatively, this could be the name of a PIF file (in either the filesystem or in a persistent data base).

In addition to these slots for each `sim-node`, other slots have been added to support the activities of the Simulation Manager. One of these is a `label` slot for the *unique* naming and identification of nodes in a tree. The label for a node is a list of integers (c_1 c_2 c_3 ... c_n) summarizing the position of the node within the tree: the node is the c_nth node in the c_{n-1}th branch of the c_{n-2}th branch, and so on, of the c_1st branch of the root node in the simulation tree. Given a node's label, and given any other node in the tree (usually the root node), it is then possible to traverse the tree to locate the specified node. Additionally, since node numbering is unique, a hash table can be constructed for fast access of a node given its label. The length of the node label indicates the *depth* of the node in the tree.

The simulation tree is constructed during the translation of a process for some particular cross section, and target simulator code is inserted into leaf sim-nodes. To aid in debugging, these codes statements often are nothing more than comments to the target simulator. For example, when a photolithography step is translated, comments are emitted to indicate that positive or negative resist has been added, exposed, developed, baked, or removed from the cross section. Steps that are masked by resist, including etches and ion implantations, are also translated to comments. As a result, appreciable fractions of a simulation tree may contain statements that do not require simulation. The `sim-node` object includes a `shares-previous` slot that points to the simulation results of a previous step for such cases. After translation is

completed the tree is traversed, comment code for the target simulator is detected, and **shares-previous** slots are set to point to the last previous leaf sim-node that contains non-comment statements (and thus contains important simulation results).

The simulation tree is a powerful data structure used to support the management of process simulation. An interesting aspect of the simulation tree is that it can be used on pre-existing Suprem-III input files. A Suprem-III input file can be loaded to create a simulation tree that has a root node with as many children as there are statements in the input file. All of the management functions described later in this chapter are then available (including incremental simulation and examination of results). The use of PFR rather than Suprem-III to describe the process has many advantages (such as hierarchical descriptions and interfaces to additional PFR-based tools), however, and is much preferable.

7.3.4 Multiple Cross Sections

An especially onerous aspect of one-dimensional process simulation is the management of multiple cross sections. It can be difficult to maintain the consistency of multiple input files each representing the process "seen by" some particular cross section, especially when the process is modified rapidly and repeatedly during design. Furthermore, separate and multiple input files may not take advantage of the sharing of simulation results between identical subsets of the process. The Simulation Manager addresses this problem in two ways. First, a single unified PFR is translated differently depending on the cross section being translated. Second, opportunities and mechanisms for sharing are provided by the Manager and th simulation tree data structure.

Since the primary mechanism for differentiation in planar technologies is photolithography, one finds that cross sections are *split* by some masking step. Each masking step potentially doubles the possible number of different cross sections on a wafer, since some regions may be exposed and others not. In actuality, two phenom-

ena act to reduce the number of different cross sections that are of interest. First, the geometry of actual layouts is such that many combinations of layout layers never occur; often the intent of layout design rules is to specifically prevent such possibilities from occurring. Second, the designer is often only concerned with the effects in some limited number of cross sections. Thus, rather than considering 2^m (where m is the number of masking steps) different cross sections, the simulation manager only manages an explicitly specified list of defined cross sections.

A graph that shows the splitting of the wafer into different cross-sections by masking steps appears in Figure 7.4. An initial implementation of the Simulation Manager

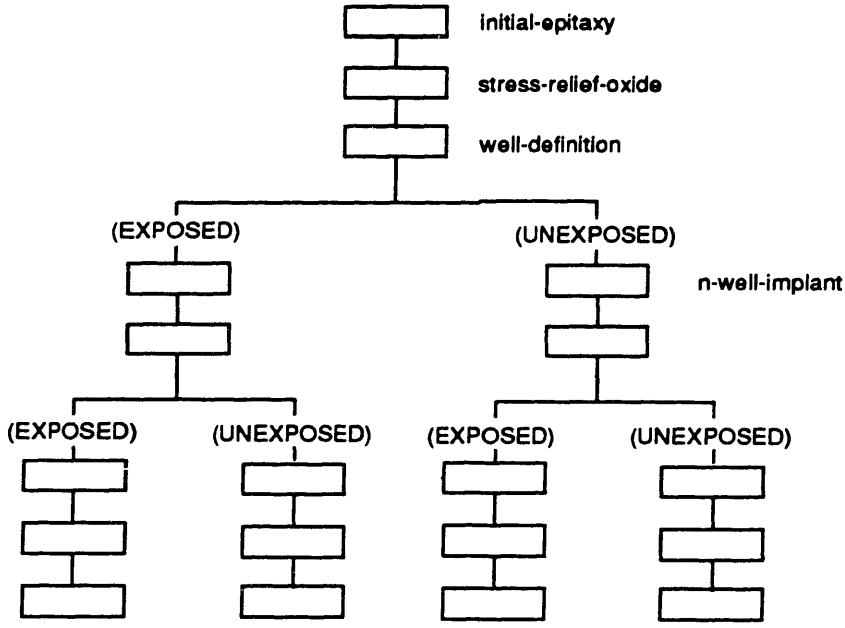


Figure 7.4: Splitting of cross sections by masking steps.

addressed the sharing of cross sections simulations based on a data structure shown in Figure 7.5. A *single* translation of the PFR into such a data structure was performed, and each process step was converted into two different code fragments, one corresponding to a "mask closed" cross section and one to an "mask open" section (if a lithography step is taking place). The definition of a cross section is then the sequence of "open or closed" lithography options for that cross section (this is the

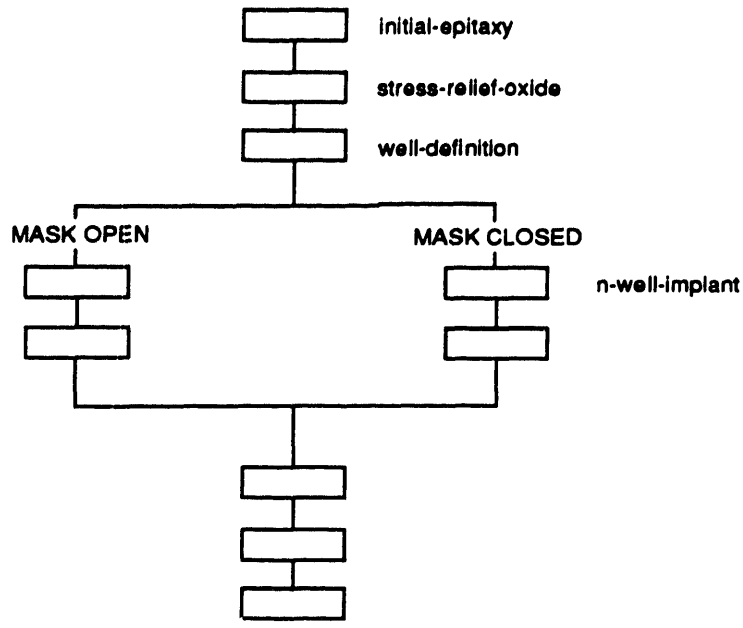


Figure 7.5: Alternative data structure for process translation.

definition proposed in the MASTIF system [3]). The generation of Suprem-III code for some cross section then corresponds to a downward traversal of the graph, where the choice of branch is made depending on the open or closed determination. The conclusions based on that implementation are (1) the open-closed definition of a cross section does not correspond well to the detailed description of processing; (2) the mechanism is not well suited to the handling of double layer resists (the fundamental splitting occurs when resist is exposed, rather than when resist is present or not); and (3) the only opportunities for simulation results sharing that this supports is up to some branch point.

The current implementation of the Simulation Manager takes a different approach. For each cross section to be simulated, the Simulation Manager generates a separate simulation tree containing target simulator statements that describe the process from the perspective of that cross section. A node-by-node comparison of simulation trees is then performed (depth-first, left-to-right to correspond to the process sequence). So long as the structure of the two trees is identical (which is always true for two sec-

tions generated from the same process description), and so long as the non-comment Suprem-III code fragments are identical, the two trees are identical and can share simulation results. Each `sim-node` object contains a `shares-parallel` slot which points to a "parallel" node in the simulation tree where the shared simulation results will be kept. This same mechanism can be used to detect and share simulation results when the original PFR changes.

7.3.5 Resimulation Minimization

If a change is made to the PFR, it is desirable to only resimulate those process steps that have changed. The Simulation Manager with its forest of linked simulation trees described above provides an efficient mechanisms for the minimization of resimulation. When the PFR for the process changes, the process is retranslated and a new simulation tree generated. The old tree and the new can then be compared exactly as described above, and the new tree can share the simulation results with the old tree. It is possible to maintain an arbitrary number of previous simulations resulting from different parameters during process design. In the current implementation of the Manager, only the previous simulation results are retained.

7.4 Management Capabilities

The Simulation Manager supports a number of management capabilities that help simplify the tasks involved in simulating and evaluating a process. In addition to the automatic process translation and (re)simulation minimization mechanisms provided above, the manager provides an interactive interface to give the user incremental control over the simulation. The designer may interactively specify the points at which the simulated wafer is to be saved, direct simulation to proceed up to some point in the process, and examine the results at points within the process. The "simulation matrix screen" (shown in Figure 7.6) is one of several Fabform screens in the interactive

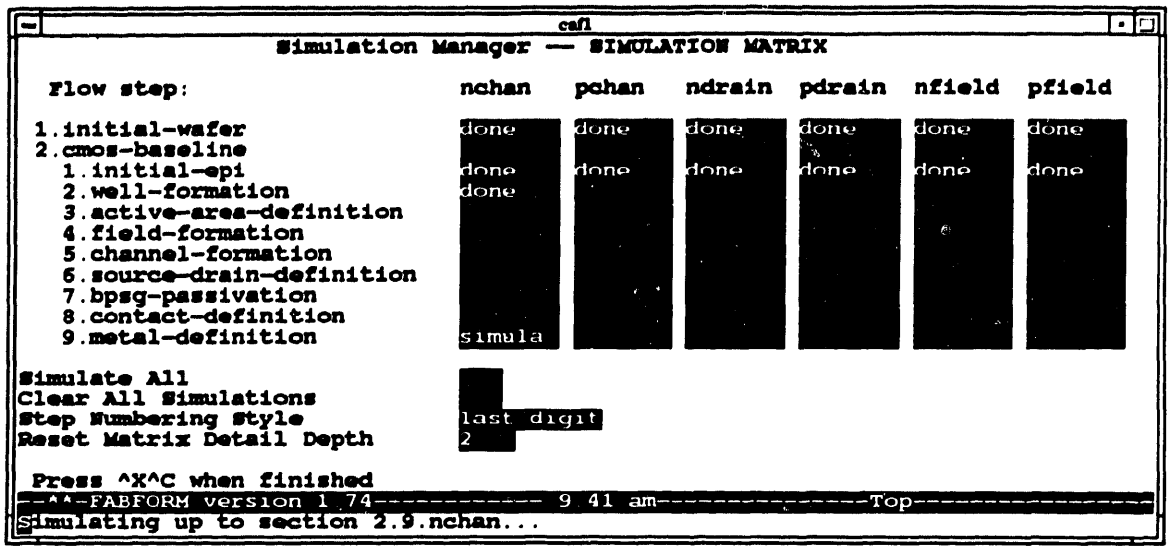


Figure 7.6: The interactive simulation matrix screen within the Simulation Manager.

interface provided by the Simulation Manager. Material layer information, electrical calculations (sheet-resistance and threshold voltages), and graphical plots may be requested from this or other screens in the interface.

The Simulation Manager encapsulates not only the knowledge about how to translate the PFR to some target simulator, but also the mechanisms for simulator preparation, start-up, and data extraction in a networked environment. In the current implementation, the Simulation Manager runs within CAFE on a Sun-3, executes simulations on a Microvax, and displays results on a desktop X terminal. The remote job execution mechanisms in the Manager are currently ad-hoc in nature, and more consistent simulation tool encapsulation approaches in a tool framework approach should be used in the future.

7.5 Conclusions

A Simulation Manager that provides an interface between the Process Flow Representation and process simulation tools has been implemented. The Manager first

translates PFR statements into data structures that support incremental, interactive simulation of multiple cross sections. When modeling of process steps is not performed by the simulator itself, it may be necessary to handle those steps within the Manager (*i.e.*, to model the effects on the wafer internally). While such "internal simulation" is possible (for example, the Simulation Manager handles lithographic steps where Suprem-III does not), a more effective and efficient approach may be to enhance the target simulator itself. In addition to process translation, the interactive *management* of simulation is essential in supporting process design. Areas for future research include the extension of the Simulation Manager to provide additional support for process synthesis (integration with the Process Advisors), design space exploration, and design optimization.

Chapter 8

PFR to SIMPL-2 Translator

A translator from the Process Flow Representation (PFR) to the input needed by the SIMPL-2 process simulator [81] has been implemented to provide two-dimensional simulation based on the PFR. A discussion of the capability provided by SIMPL-2 in the context of the generic process model appears in Section 8.1. The implementation of the translator is summarized in Section 8.2. The translation of PFR information, including the estimation of implant diffusions and special handling for double-layer resist processing, are discussed in Section 8.3. Issues involved in interfacing to SIMPL-2 via the Simulation Manager are discussed in Section 8.4. Finally, conclusions are offered in Section 8.5, including recommendations for modifications to SIMPL-2.

8.1 Change in Wafer State Simulation

SIMPL-2 differs markedly from the Suprem-III process simulator in two ways. It does not simulate physical processing in the detail normally associated with process simulators. For example, the diffusion of impurities in silicon, which is the core of such simulators as Suprem, is not handled. Furthermore, SIMPL-2 is well-coupled to layout information, so that one can correspond two-dimensional profile information to the layouts that produced that profile. Thus SIMPL-2 focuses on providing an

essential connection between circuit/device layout and resulting structure rather than on the modeling of physical processes.

These features of SIMPL-2 are interesting in the context of this thesis from two related points of view. First, the simulator operates on descriptions of the process at the *change in wafer state* level, compared to such treatment-level simulators as Suprem-III. SIMPL-2 is thus appropriate for use in very early stages of process design (where one is working to "sketch-in" the basic process sequence). Second, SIMPL-2 is a two-dimensional simulator, and is useful to test the ability of the PFR and the Simulation Manager to deal with two-dimensional aspects of semiconductor processes.

8.2 Implementation

The PFR to SIMPL-2 translator has been implemented similarly to the Suprem-III translator (Chapter 7). In this case, however, the translation is not dependent upon the cross section definition, so that only one translation is needed for multiple cross sections. The cross section cut-line is maintained using the usual SIMPL-2 mechanisms. As in Suprem-III translator, a simulation tree is produced as a result of the translation, where information is entirely drawn from the :change-wafer-state attribute information in the PFR. (An exception to this is future diffusion estimation, discussed in Section 8.3.2.)

It is important that one be able to establish and understand a correspondence between intermediate wafer structures produced by the SIMPL-2 simulator with the results produced by the Suprem-III interface. More generally, it is desirable that one be able to look at some step in the middle of the process in different ways. The mechanism used in this work is to use the operation decomposition tree as the structure that binds all process information together. Both the SIMPL-2 and Suprem-III translators produce simulation trees that are essentially isomorphic to the operation tree, and hence isomorphic to each other as well. It is therefore easy to correspond

intermediate results produced in different ways (*e.g.*, the treatment level simulation of a step with its change-wafer-state level simulation).

8.3 Translation Issues

8.3.1 Double Layer Resists

The handling of lithography within the SIMPL-2 simulator is much more complete than that within the Suprem-III simulator. While Suprem-III does not intrinsically handle lithography†, the modeling of lithography is a crucial component of SIMPL-2. Despite this focus, special handling of lithography by the PFR translator remains necessary.

The root of the problem is the inability of the SIMPL-2 simulator to directly model the bake operation. While it might seem surprising that such a minor operation is the cause of the difficulty, the following double resist example illustrates the resulting limitation. In a double level process, the following steps occur:

1. Photoresist deposition. The state of the resist is :positive-resist.
2. Exposure. Some part of the resist state on the surface of the wafer is changed to be :exposed-positive-resist.
3. Develop. All :exposed-positive-resist is removed from the wafer.
4. Bake. All :positive-resist changes state to :baked-resist.
5. Photoresist deposition. More photoresist is added to the wafer. Now there is *both* :positive-resist and :baked-resist on the wafer.
6. Exposure. Some part of the :positive-resist is exposed, but the baked resist does not change state. There are now three states of resist on the wafer (:baked-resist, :positive-resist, and :exposed-positive-resist).
7. Develop. The :exposed-positive-resist is again removed from the wafer.

†A substantially modified version of the 8628 Stanford University distribution of Suprem-III has been used in this work; recent versions by third-party vendors have been modified to better handle lithography.

8. Bake. The :positive-resist is converted to :baked-resist.
9. Processing. The selected processing, such as implantation, occurs.
10. Etch. All baked resist is removed.

Simpl-2 provides mechanisms for all of the above steps *except* the bake step, where the state of a resist is changed. The two-level resist process depends on the difference between baked positive photoresist and newly deposited positive resist (only the latter is sensitive to light). The SIMPL-2 exposure changes the state of the resist that is exposed to light, but does not enable one to change the state (that is, the name) of material layers on the wafer. If only a single SIMPL-2 material type is used for resists (such as "RST"), then it it impossible to distinguish between the baked resist and the "fresh" resist.

The following solution is used in the SIMPL-2 translator. For the wafer undergoing processing, the set of resist states on the wafer is maintained by the translator. During a resist deposition, if prior resist already exists on the wafer, then a new layer is used ("RST2"), which, when exposed, is converted to "ERS2". The :deposit, :expose, :develop, :bake, and :etch steps must all modify the resist and wafer states appropriately. This implementation essentially models the :bake operation entirely internally, and does not use SIMPL to keep track of baked resist states. A SIMPL-2 cross section produced by the PFR Simpl-2 translator is shown in Figure 8.1, where both layers of resist can be identified.

An alternative implementation is to model the :bake using the external mechanisms provided by SIMPL-2. The problem is to convert unexposed :positive-resist to :baked-resist during the bake operation. The exposure operation can be used to fictitiously "expose" all "RST" material to "BRST" material. This requires that a mask that is always clear (or dark and then inverted) is used. This has the disadvantage that such a fictitious mask must be added to each and every layout.

A much more attractive alternative would be the enhancement of SIMPL-2 to handle the bake operation. At the simplest level of modeling, this could be imple-

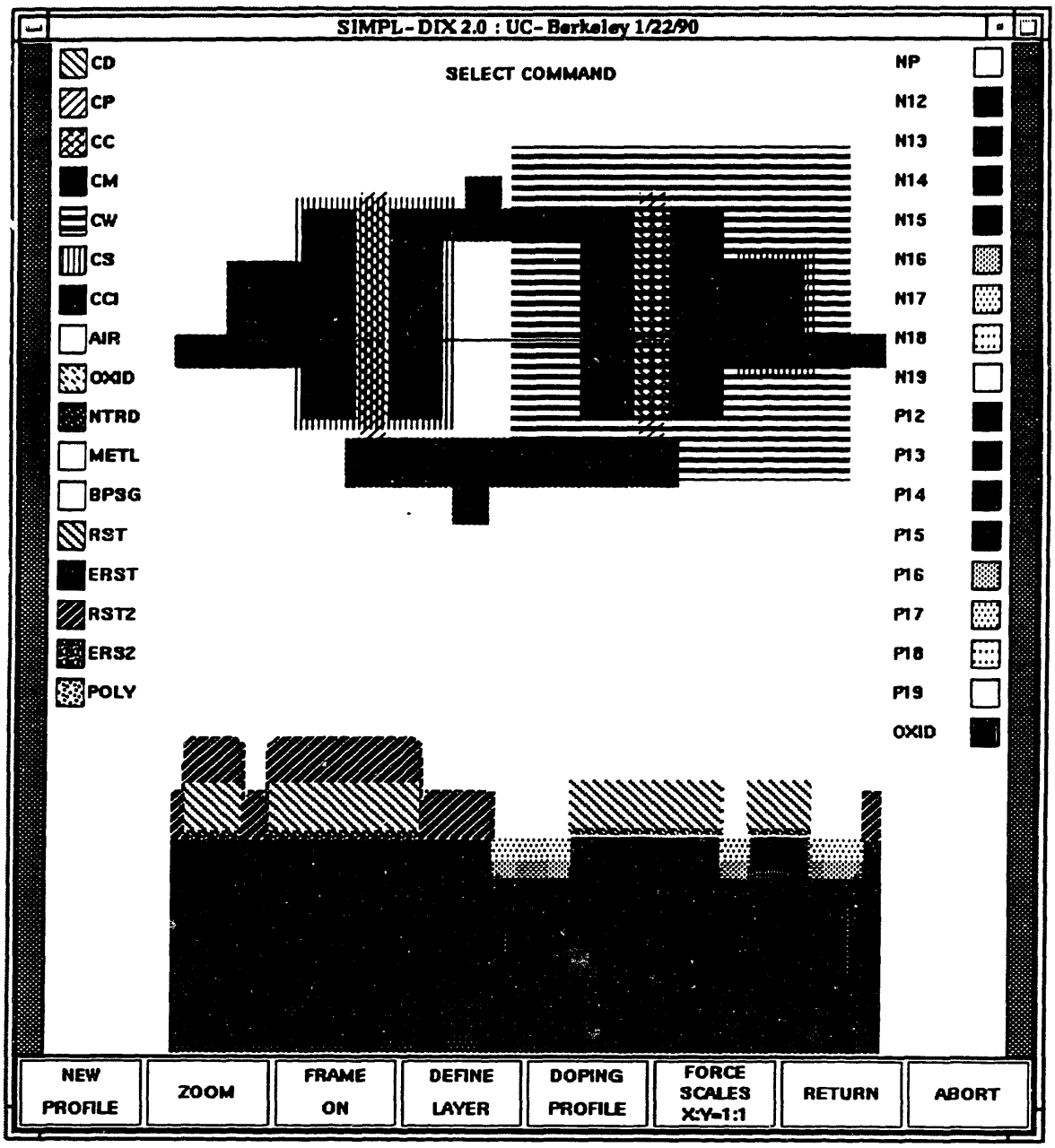


Figure 8.1: SIMPL-2 simulation of the double layer resist processing. The step shown is right after the p-field threshold adjust ion implant in the MIT CMOS baseline process. The resist from the active area definition is still on the wafer, in addition to the n-field cover resist.

mented simply by allowing one to change the name of some layer on the wafer. The modeling of the bake operation could also be extended to include a factor to model the shrinkage of resists during baking.

8.3.2 Diffusion Estimation

A more serious limitation in the SIMPL-2 simulator is the lack of a diffusion operator. It is not possible, for example, to implant an impurity with shape factors that correspond to an individual step, and have the implant then diffuse during subsequent steps. This is a severe limitation, and complicates the incremental description of process steps. If diffusion of implants is ignored, extremely poor results for the doping profiles are experienced, even to the degree of crude graphical depiction. In Figure 8.2, a SIMPL-2 simulation of the well-formation in the MIT CMOS baseline process is shown at the end of the well-formation processing (including the subsequent oxidizing drive-in of the implanted twin wells). The implants do not reflect the drive-in, and the wells do not appear at all (the implanted profiles with their normal straggles have been absorbed into the oxide during the drive-in).

It is possible to obtain better results by estimating the total amount of diffusion that the implant will experience in future processing, and implant the profile so as to include the effects of this future diffusion. The PFR translator provides the opportunity to automate this aspect of implant diffusion modeling. For any implant, the "future" diffusion-time product Dt_{fut}, where D is diffusivity and t is time, can be calculated by examination of the PFR. The implanted profile is then modified to reflect this future diffusion with an equivalent straggle ΔR_p^*:

$$\Delta R_p^* = \Delta R_p + \sqrt{2Dt_{fut}} \tag{8.1}$$

where ΔR_p is the straggle of the original implant.

The Dt_{fut} sum is found as follows. For a node n (which has parent p), the future

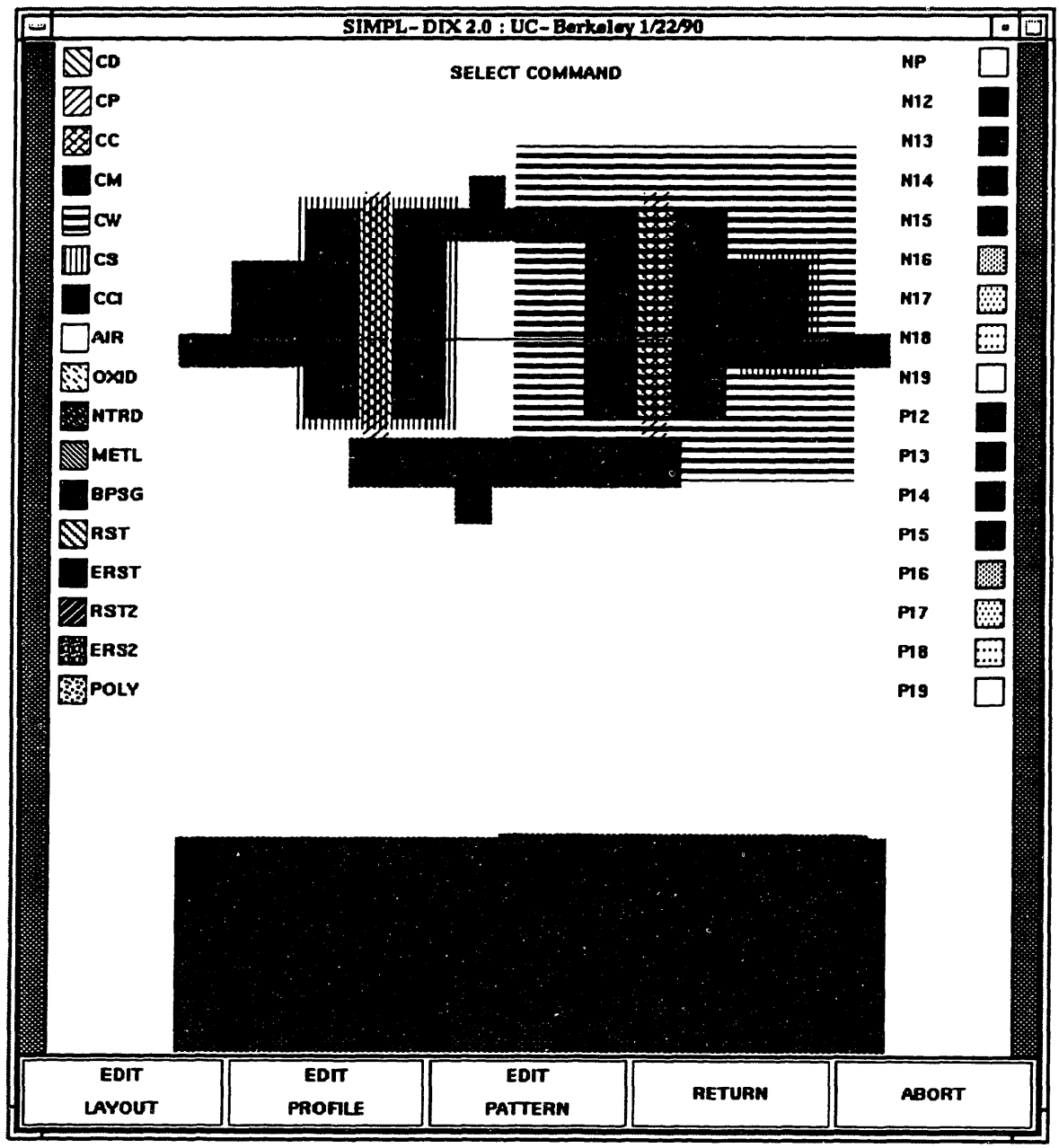


Figure 8.2: SIMPL-2 simulation of the well-formation subprocess, without estimates of the diffusion that the well implants will experience in the future. As a result, the background epi doping is all that is seen.

diffusion-time product is given by:

$$Dt_{fut}(n) = Dt_{sub}(p) + Dtfut(p) - \sum_{s} Dt_{sub}(s) \tag{8.2}$$

where s are the previous siblings of n (sub-operation of p occurring before n). Note that it is important that only parent and previous sibling nodes are examined, because future sibling nodes are not yet created during the incremental translation of the PFR. The Dt_{sub} for some node represents the diffusion-time product experienced by the node itself (and does not include any "future" processing due to the node being the child of some parent node):

$$Dt_{sub}(n) = Dt(n) + \sum_{c} Dt_{sub}(c) \tag{8.3}$$

where c are the children of node n. The $Dt(n)$ is incremented whenever a thermal processing step represented by `:thermal` treatment primitives is encountered in a node.

This estimation of the "future" diffusion an implant experiences is quite successful. Figure 8.3 shows a simulation where the subsequent thermal processing has been included in the SIMPL-2 code produced by the translator. In this case the twin wells of the CMOS baseline process do show up (though in black and white, the stipple patterns of the approximately equivalent doping levels appear similar, and only the interface between the wells is clearly depicted). The cost of the future diffusion estimate has qualitatively involved a ten-fold increase in the time it takes to translate the process. For the complete CMOS process, the difference was 583 cpu seconds with diffusion estimates, compared to 53 seconds without. This compares with the approximately twenty minutes required by SIMPL-2 to process the resulting input file.

8.4 Simulation Manager Interface

The SIMPL-2 translator has not yet been incorporated into the Simulation Manager interface. This is clearly necessary in order for the non-LISP user to invoke the

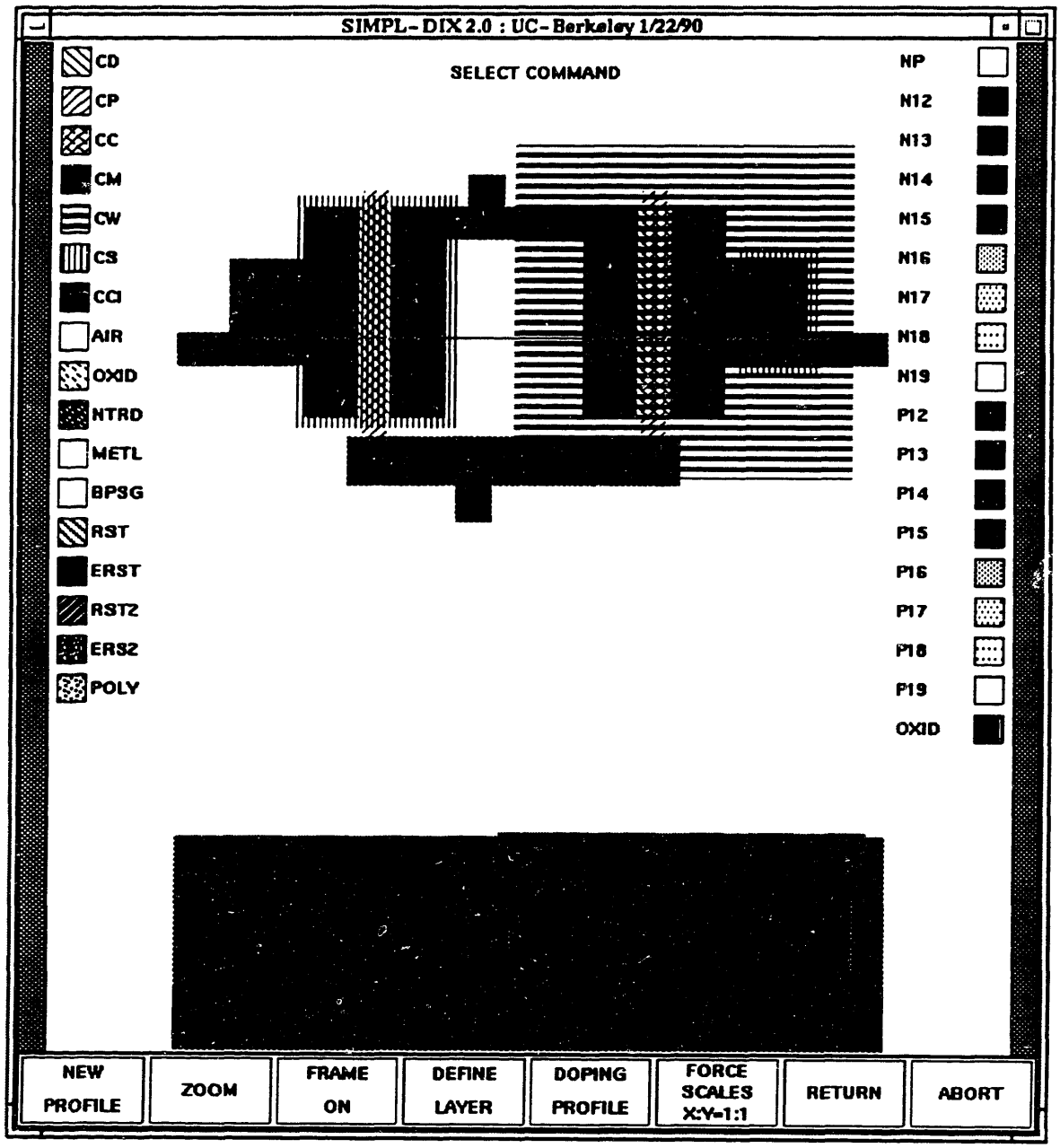


Figure 8.3: SIMPL-2 Simulation of the well-formation subprocess, including estimates of the diffusion that the well implants will experience during the well drive. Here, the n and p wells are apparent.

translator, start the SIMPL-2 simulator, and view the results using a common, PFR-based interface.

Two modifications to the SIMPL-2 program would aid in the simulation of fabrication processes and the integration of the SIMPL-2 program into a complete TCAD system. First, the simulator can be trivially enhanced to include a "convert" (for a bake) operation that converts one material type to another. Second, extension of the SIMPL-2 command line arguments so that the program can be used in a "viewer" mode when called from a UNIX shell would increase the program's flexibility.

8.5 Conclusions

An interface from the Process Flow Representation to the SIMPL-2 process simulation has been implemented. This interface provides the ability to perform two-dimensional simulations of change in wafer state process descriptions written in the PFR. Limitations in the ability of SIMPL-2 to model complete fabrication processes complicates the translator. The translator extends SIMPL-2 to describe implant diffusions and to handle double layer resist processing.

Chapter 9

Process Advisors

Most tools for semiconductor process design have focused on accurate *simulation* of processes. Given a description of a starting wafer and a fabrication process, tools such as Suprem-III [24], Suprem-IV [25, 70], Sample [27, 28], Simpl-2 [82], or BICEPS [83] produce descriptions of the resulting wafer state. Very often, however, this is not the task facing the process designer. Instead, the designer has in mind a starting and an ending wafer state, and needs to formulate a process that accomplishes the necessary modification or mutation of the wafer. That is, the designer's task is to *synthesize* a treatment or other description of the step.

The need for tools to aid in synthesis in addition to simulation was recognized in MASTIF [3]. This chapter discusses work done to directly address the synthesis problem. Tools called *Process Advisors* have been designed and developed to aid in the synthesis of *treatment* information based on *change in wafer state* goals. This chapter introduces these tools, describes their user interfaces, and discusses their impact on process design.

---

†This work was done in collaboration with Partha Saha [31] and Denis Akkus [32].

9.1 The Problem: Treatment Synthesis

The intent of the Process Advisors is to help in treatment synthesis. Two qualitatively different cases in which one must determine treatment parameters can be identified. In one case, the designer must determine the parameters for a unit process step based on the wafer structure at the end of that particular step; this is defined as *unit process synthesis*. An example is the determination of parameters to grow a gate oxide of a desired thickness. In the second case, the determination of treatment parameters depends also on *downstream* requirements, such as the state of the wafer at the end of all processing. An example of this kind of synthesis is the determination of implant parameters such that after all subsequent diffusion steps a particular junction depth is achieved. The Process Advisors have been restricted to unit process synthesis only. Three unit processes are considered: ion implantation, oxidation, and diffusion in silicon.

9.2 Approach: Fast Analytic Estimates

The philosophy behind the Process Advisors is to provide help in the most basic of activities that process designers must perform time and time again. Determination of oxidation, diffusion, and ion implantation parameters are such key activities. Two kinds of common subtasks occur: (1) find an initial guess for some parameter; and (2) enter into a simulate-compare-revise loop (or a fabrication-compare-revise loop) to improve the process parameter as necessary. A goal of the Advisors is to both suggest an initial guess, and then to suggest revisions to that guess based on simulated or measured data.

The architecture of the Process Advisors is shown in Figure 9.1. The separation between advisor internals and user interface is important. The implementation of the Process Advisors internals has been performed and reported by Partha Saha [31]. Described here are the interfaces a user of the tool sees for each advisor and a summary

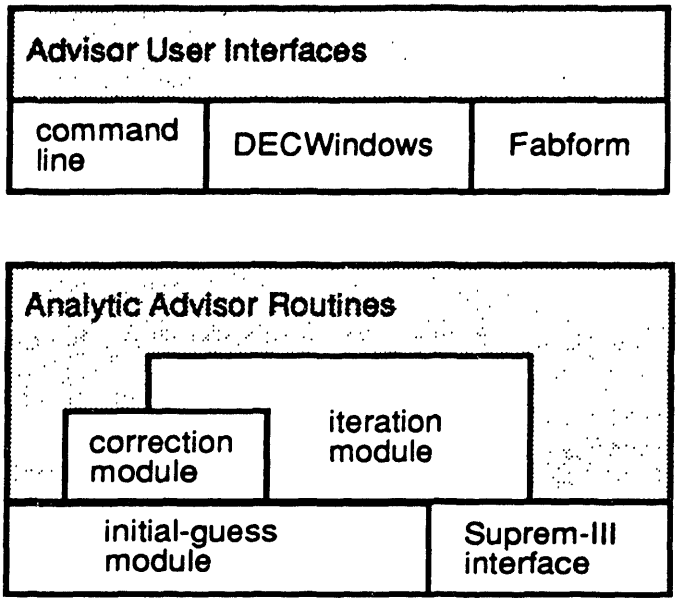


Figure 9.1: The implementation architecture of the Process Advisors. User interfaces and calculation routines are implemented for oxidation, diffusion, and implantation unit process steps.

of the models and solution approaches used within the advisor.

9.3 User Interfaces

9.3.1 Program Interface

Three different user interfaces are provided by each advisor. The first and most essential of these is a programmatic or functional interface. Each of the Advisors provides functions to perform the following: calculation of an initial guess for some process parameter, calculation of a revised guess based on simulated or measured results, and analytical calculation (or "simulation") of the effect produced by a postulated process parameter.

```
#!/bin/csh
#
echo .col conc time
foreach conc ( 1e14 1e15 1e16 1e17 1e18 1e19 1e20 1e21 )
  set time = `oxidation dry bg_imp=boron final_ox=.0300 \
                sf_imp=phosph sf_conc=$conc temp=950 \
                action=initial`
  echo $conc $time
end
```

Figure 9.2: A script for producing tabular information about required oxidation time as a function of silicon surface concentration.

9.3.2 Command-Line Interface

Second, a command-line or shell-level interface is implemented on top of these program interfaces. The shell-level interface is useful as a basic building block for use both directly by the user, as, for example, when he or she does not have a bit-mapped display available to use the graphical interface. The shell-level interface is ideal for use in constructing more powerful shell scripts (*e.g.*, to iterates over some range of processing parameters in order to relate those parameters to analytic results and tabular data). A simple shell script to relate the time needed to grow 300 Å of oxide depending on the surface concentration is shown in Figure 9.2; the resulting output tabular data is shown in Figure 9.3.

Important requirements on the shell level interface are that it provide mechanisms to specify directly all of the possible goals and parameters (the interface should not query the user for responses), as well as mechanisms for reporting results in a form that can conveniently be interpreted and used by the shell script. In this case, the output of advisor information is to *stdout*.

```
.col conc time
1e14 92.3764
1e15 92.376
1e16 92.3727
1e17 92.3392
1e18 91.9816
1e19 85.4058
1e20 34.9003
1e21 6.15232
```

Figure 9.3: Tabular data (suitable for use in graphical plotting programs) relating oxidation time to grow 300 Å oxide as a function of surface concentration, produced by the script of Figure 9.2.

9.3.3 Interactive Interface

The program and shell interfaces are useful as basic components for both application programmers and end users to use in building up higher-level utilities. In addition, however, interactive interfaces are especially important for the occasional user. One such interactive interface was implemented using the Fabform ASCII-text, template-based interface toolkit [12]. This interface was subsequently replaced by a high-quality graphical interface that a user may intuitively use. By using the DECwindows (X11) toolkit with its easily understood widgets including buttons, toggles, choice menus, and so on, users of this Process Advisor interface have less difficulty understanding how to use the Advisors.

9.4 Advisor Summaries

9.4.1 Oxidation Advisor

The oxidation advisor is illustrated in Figure 9.4. The oxidation advisor takes as input an initial wafer state having an initial oxide thickness over silicon with a uniform background doping (and some orientation), and with some additional surface dopant

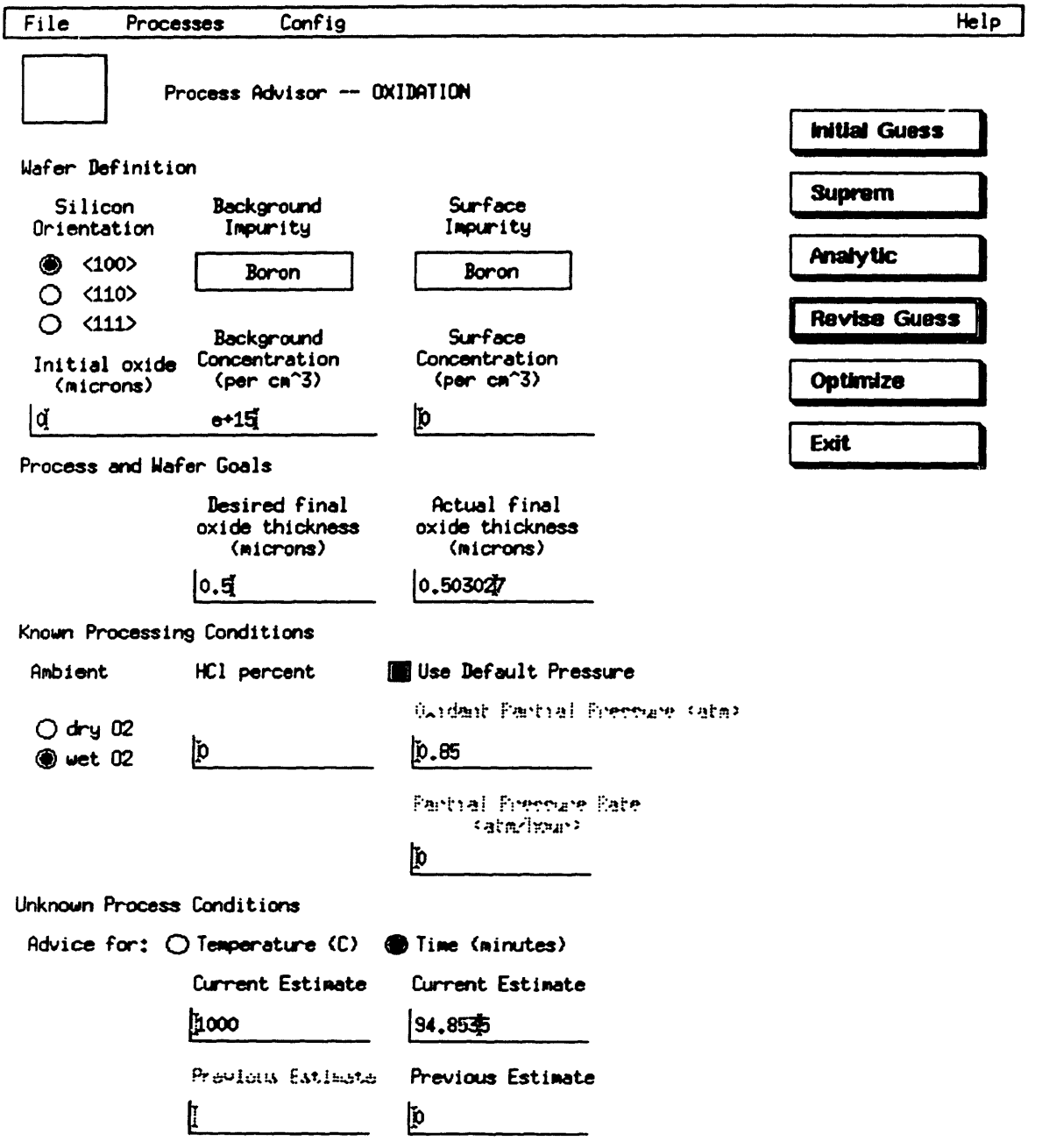


Figure 9.4: The oxidation advisor.

near the interface. The wafer goal is a final oxide thickness. Processing conditions that must be specified include the ambient (dry or wet oxygen), the HCl percent, the partial pressure of oxidant, and the oxidation temperature. The advisor solves for the time at these conditions needed to achieve the desired oxide thickness.

A modified Deal-Grove model [26, 84] is used for the initial guess calculation. With the addition of parameters for the effects of HCl, substrate doping, and pressure, very good prediction of oxide thicknesses is possible via the oxidation equations. The equations are solved using a simple iteration loop to give nearly instantaneous predictions for the oxidation time. Corrections to the linear rate constant are made to produce revised guesses.

9.4.2 Diffusion Advisor

The diffusion advisor is illustrated in Figure 9.5. The initial wafer state is a uniformly doped silicon wafer with a specified background dopant species, dopant concentration, and orientation. In addition, an indicated impurity will be present at the surface of the wafer to act as the diffused dopant. The wafer goal is a desired final junction depth. Three types of diffusion conditions may be solved for: a constant source, limited source, or two-step diffusion, with processing conditions that must be specified depending on which type of doping process is to take place. In this version, the diffusion temperature must also be specified. The time necessary to achieve the desired junction depth is produced as a result.

Well known analytic models for diffusion are used for these specialized diffusion conditions [85]. A constant diffusivity is assumed, but its value is estimated based on the overall processing conditions. This diffusivity is updated based on a simulated or measured result, and can be considered a time and spatially averaged *effective* diffusivity for the process step. Analytic "simulation" is also carried out based on the simplified initial conditions, though some work has been done to handle arbitrary initial profiles using a superposition of Gaussian profiles for which individual analytic

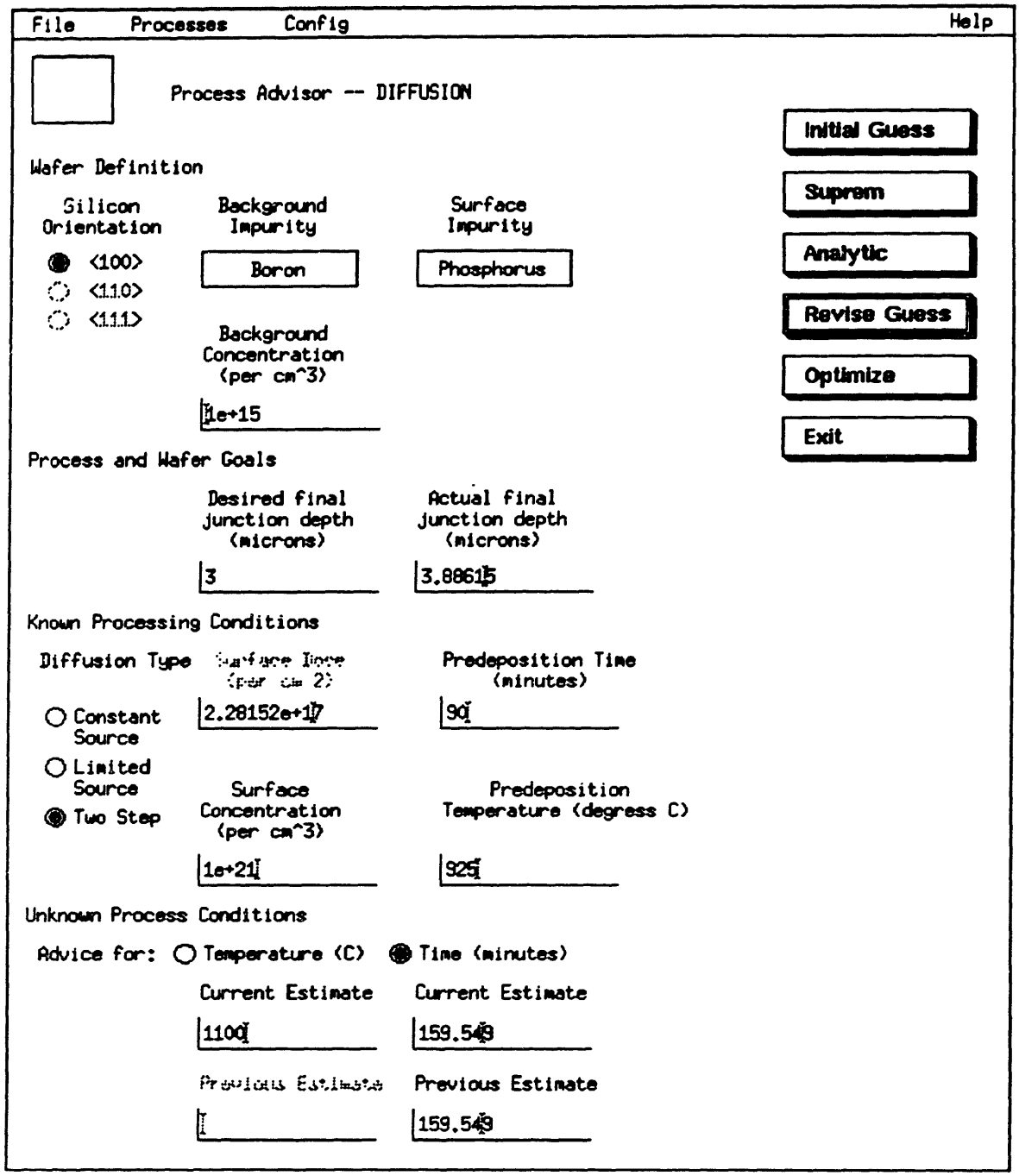


Figure 9.5: The diffusion advisor.

solutions are possible [31, 32].

9.4.3 Implantation Advisor

The implantation advisor is illustrated in Figure 9.6. Here the initial wafer is some uniformly doped silicon wafer with a specified doping concentration. The wafer goal is an implanted junction depth, and the required parameter is the implant energy. The advisor solves for the implant dose necessary to achieve the specified junction depth. The implant advisor as defined here has proven to be less useful than the other Advisors, because a designer rarely is implanting for these wafer goals and with a known energy. Rather, it is often the energy that is the parameter that must be found to achieve a desired implant peak depth or junction depth. The implant advisor has been implemented using the same tabular implant tables and Gaussian and Pearson-IV implant profile models as in Suprem-III [24].

9.5 Discussion

9.5.1 Accuracy and Convergence

The accuracy of the initial guesses (with respect to numerical simulation via Suprem-III) depend upon the particular advisor and the step parameters. Oxidation time estimation tends to be very accurate across a broad range of conditions (typically within 5-10%), and convergence to less than 1% difference usually requires only a single additional iteration. Diffusion estimation is good in low concentration intrinsic conditions. Thermal doping processes, however, tend to involve high concentration diffusion, and the single "lumped" constant diffusivity is of limited accuracy. Initial estimate errors can be on the order of 20–50%. Iteration tends to converge fairly rapidly, typically requiring 3 to 4 simulation iterations.

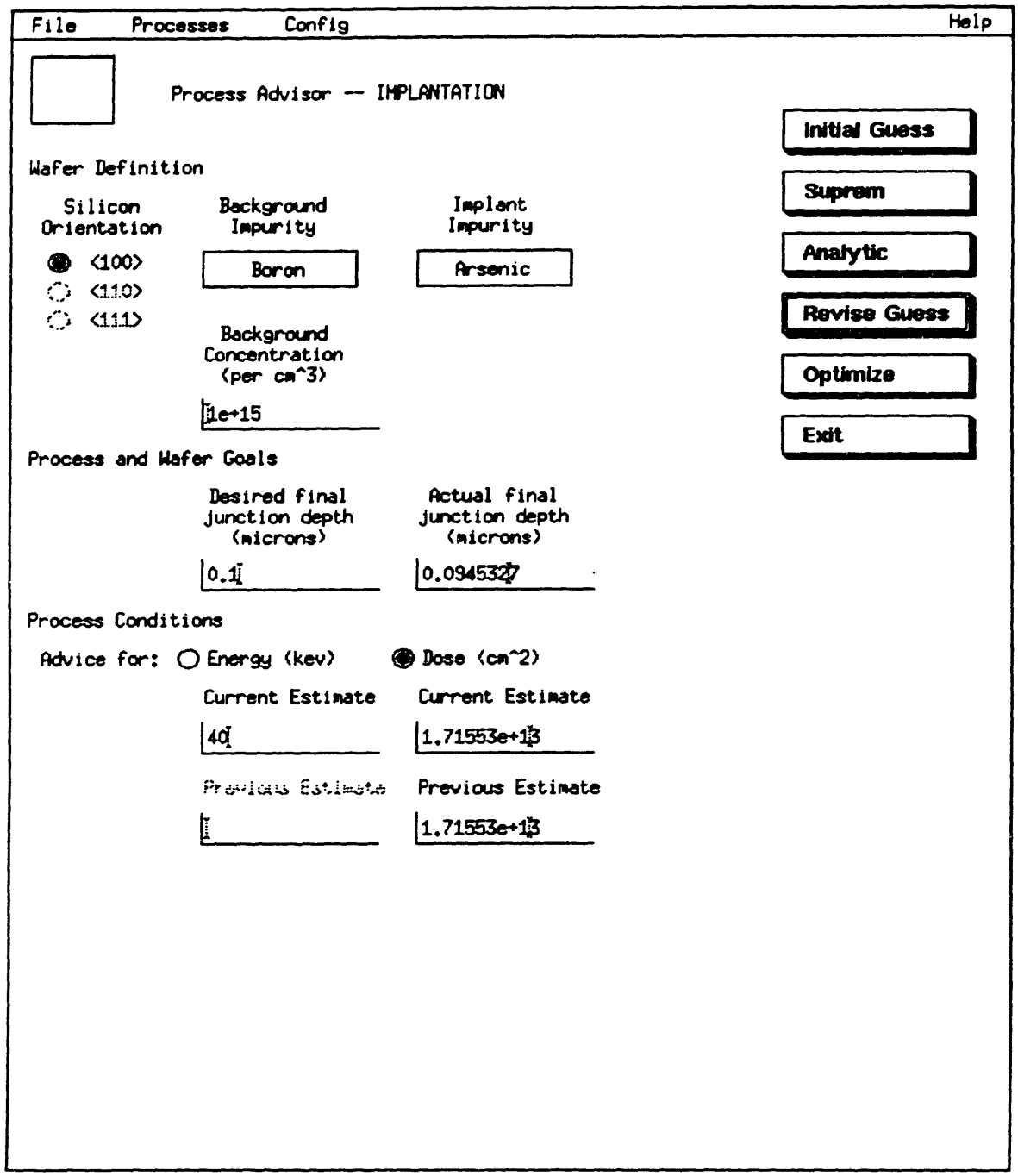


Figure 9.6: The implantation advisor.

9.5.2 Ease-of-Use

The experience of students within the MIT Integrated Circuits Laboratory who have used the oxidation and diffusion Advisors has been very positive. Students having little to no experience with numerical simulators such as Suprem-III find the Advisors extremely helpful and intuitive to use. Other students tend to use the Process Advisors as a precursor to extensive simulation, and find that they can substantially narrow the range and number of simulation experiments that they must perform. Finally, the Advisors are helpful in a surprising pedagogical fashion: intuition about the relationships between process parameters and wafer conditions can quickly be established via the tools (*e.g.*, the dependence of oxidation time and thickness on surface doping as in Figure 9.2).

9.5.3 Relationship to Optimization

Optimization is a general and flexible way of determining process parameters. The primary advantage of numerical optimization when compared to direct analytic solutions is that knowledge about the models is not necessary, and general purpose optimization techniques can be applied. Furthermore, such techniques remain applicable even when the analytic approaches break down due to the complexity of the structures and processes, so that numerical optimization can be applied to the determination of multiple treatment parameters based on *downstream* goals. The application of numerical optimization techniques coupled with process simulation for the evaluation of the structures has been demonstrated by other workers [30]. The primary disadvantage of such approaches is that numerical simulation is usually required, so that fast analytic estimates like those provided by the Process Advisors remain important.

9.5.4 Expert Systems for Process Synthesis

The use of expert-system technology for the synthesis of recipe information has also been demonstrated [18]. Such work is close in spirit to the Process Advisors in that mechanisms besides numerical simulation and optimization are used. In the determination of polysilicon recipes by the BIPS system [18], two steps are performed that correspond roughly to the two-stage process model discussed in Section 4.8. First, an "equipment independent" recipe is found that achieves a number of goals including polysilicon layer thickness and resistivity. Second, this recipe is "compiled" or "translated" into the settings for specific pieces of equipment. The intermediate recipe includes a description of the treatment that the wafer sees (*e.g.*, temperature and time profiles) in addition to generic equipment steps such as wafer pushes and pulls. The intermediate recipe is found not by numerical simulation and optimization, but rather by a combination of rule-based, analytic equation solution, and empirical data interpolation methods.

The Process Advisors differ in several ways from the BIPS approach. First, the Advisors focus only on the treatment synthesis step, and do not attempt machine dependent recipe generation. By separating out the two parts, the Process Advisors are appropriate tools at very early stages in process design, where the full power and complexity of the BIPS systems is unnecessary. Second, the Advisors strictly use analytic model equations for suggesting values. Finally, the Advisors make suggestions not only for initial operating points, but also use the same models to suggest ways to revise guesses of treatment parameters when additional results become available. The approaches used in the Process Advisors are extremely simple, but illustrate the power of model revision based on simulated or measured data.

9.5.5 Relationship to Device Synthesis

Work analogous to the Process Advisors has been done at Stanford [86], though with a slant toward diagnosis rather than design support, and for devices rather

than processes. The general approach described here, where analytic models relating device and wafer parameters can be solved in either the forward or reverse direction, may also be applicable for use in device design.

9.5.6 Nested Models

One way to look at the role of the Process Advisors is that they provide a way to "jump-start" conventional process simulation and optimization loops. Simulation can similarly be viewed as a way to jump-start fabrication. Within the Process Advisors, other ways to jump-start the solutions are likewise used. For example, simplified analytical models may be used to estimate initial conditions for internal iterative solution loops. A general principle seems to be at work here: it is often possible to resort to simplified models in order to reduce the amount of time needed to find a solution. Such simplified solutions are appropriate so long as the total solution time is reduced.

9.6 Extensions

Several extensions of this work are possible. First, the application of analytical models to more varied or arbitrary initial wafer structures is possible, and some work toward that goal has been done [32]. Secondly, extension to the design of devices might also be possible. Clearly, the choice of solution parameters in the implant advisor is not a good one, and that advisor can be revised. Finally, the Advisors can be extended to solve for additional unknowns (both so the user can choose among a broader range of unknowns, and so that multiple unknowns can be solved for simultaneously). The approach used in the generation of revised guesses might also be applicable within numerically based optimization loops, at least in the early stages of such optimizations.

9.7 Summary

Some of the general lessons that come out of the investigation of Process Advisors are summarized below.

- Tools that do more than just simulate are necessary and valuable.
- A spectrum of models is needed for use in different phases of process design and analysis, including both numerical and analytic models (as well as empirical).
- User interfaces should include program interfaces, shell-level interfaces, and high-quality graphical user interfaces in order for the tool to be accepted and adaptable within other systems.
- Tools that are extremely fast and easy to use are necessary for quick exploration of the design space.
- An unexpected side benefit of such interaction is that the Advisors help to form an intuitive and empirical understanding of the relationships between wafer and process parameters. This suggests that the tools might be useful for educational purposes.

The Process Advisors discussed in this chapter are simple examples of direct computer assistance for use during the "synthesis" of a fabrication process. The implantation, diffusion, and oxidation unit Process Advisors aid in the synthesis of treatments for some of the most common unit process steps. The oxidation and diffusion Process Advisors have been found to be extremely useful, despite the many limitations imposed on them. Much research remains before accurate, flexible, and efficient process synthesis tools are widely available.

Chapter 10

Process Verification

To support fabrication process design, transfer of a process to manufacturing, and actual fabrication, it is often desirable to confirm conformance of a process description to various sets of design rules or requirements. Several types of *process verification* are considered in this chapter, and simple demonstrations of verification capability using the Process Flow Representation are presented.

10.1 PFR Syntactic Checks

The PFR imposes a strict syntactical structure on the description of fabrication processes. The implementation of a tool based on the PFR is greatly simplified if it can be assumed that the PFR descriptions to be manipulated by the tool are syntactically correct. Because the PFR is much like a programming language, syntactic mistakes often arise during the generation or modification of a PFR description. A program, similar to the *lint* syntax checker for the C language, is needed to help identify potential syntactic problems in PFR descriptions.

One syntactic error is an *undefined reference*, or the use of a definition (or symbol) within a PFR description that has not been defined. When encountering an undefined reference, for example, a PFR-based tool using the core flow evaluator described in

chapter 5 will simply abort with a Common Lisp error. This is clearly undesirable.

To demonstrate the possibility and usefulness of syntactic verification, an undefined-reference check has been implemented. The core evaluator `get-value` function is modified to optionally emit warnings rather than to abort to Common Lisp (via an `error` call) whenever a value is requested by the core evaluator that does not exist. The undefined-reference checker recursively evaluates and walks a specified hierarchy. Both the essential operation hierarchy (specified via `:body` attributes), and the hierarchies of other attributes within an operation (which may also be specified via function calls) are traversed. Experience with the undefined reference checker indicates that such a check can be performed relatively quickly (45 seconds clock time to check the entire CMOS baseline PFR), and that it successfully eliminates such errors during subsequent use of the PFR by other programs (*i.e.*, the simulation translators and opset/traveler generators).

An important observation results from the use and implementation of this checker. The most effective implementation approach is to enhance the core evaluator to directly support an error checking mode. This helps to assure consistency between the checker and the core evaluator, and eliminates the need to essential re-write the core-evaluator within the syntax checker. To further illustrate this approach, the core evaluator has also been enhanced to detect and report function argument mismatches, unknown special forms, and flow file location failures, during syntax checks, at the cost of less than 20 additional lines of code.

10.2 PFR Design Rule Checks

Several different kinds of processing *design rules* are possible. One set of rules is needed to check that facility safety guidelines are obeyed by a process description. Another set of rules may govern or verify the manufacturability of a structure. Still another set may describe the "schedulability" of a process. A small set of experimental

design rule checkers have been implemented to test these ideas and are described below.

10.2.1 Time-Required Checker

The time-required checker is governed by the following rule:

> It is an error (a warning is issued) if the sum of the time-required in the body of an operation is greater than the time-required specified in an operation.

The checker is implemented as a Common Lisp function that walks through a given PFR form searching for violations of this rule. The code for the time-required checker is shown in Figure 10.1. An example execution of this design rule check is shown in Figure 10.2.

This implementation of the time-required checker has the design rule embedded directly within the code of a routine that walks through the process. That is to say, there is no separate representation of the design rule which can be executed by a generic design-rule checker engine. Such an engine would ease the burden of writing design rule checkers. It could also become prohibitively expensive to check a process against a library containing a large number of such rules, unless it were possible to check these rules in parallel during a single walk of the process. A general mechanism for the representation of fabrication design rules would also contribute toward the understanding and manipulation of process knowledge, and is an interesting area for further research.

10.2.2 Resist-Diffusion Check

A checker for the following rule has also been implemented:

> No resist should be present on a wafer before a high temperature furnace step (a diffusion or oxidation).

```
;; Check BOTH at the top level, and recursively everything at a lower
;; level.  We're not really interested in what is returned here, just
;; in the warning messages printed as a side effect.
(defun time-check-interp (form)
  (let ((op-object (fl-eval form))
        (op-name (fl-name form)))
    (cond ((or (typep op-object 'flow)
               (typep op-object 'operation))
           (cond ((time-required op-object)
                  (let* ((op-time (first (fl-eval (time-required op-object))))
                         (body-time (body-time (body op-object))))
                    (if (<? op-time body-time)
                        (format t "Warning: Time-required in ~S [~A]
is less than the time-required in its body [~A]~%~%"
                                (cond (op-name op-name)
                                      (t form))
                                (time-string op-time)
                                (time-string body-time))))))
           (dolist (body-part (body op-object))
             (time-check-interp body-part)))
          ((fl-sequence? form)
           (dolist (sequence-part (fl-sequence? form))
             (time-check-interp sequence-part))))))

;; Finds the sum of the time-required specified in the given body.
(defun body-time (body)
  (cond ((null body) 0)
        (t (apply #'add (mapcar #'fl-find-time body)))))

;; fl-find-time tries to evaluate a body part to find a time-required
;; field.
(defun fl-find-time (body-part)
  (if (null body-part) (return-from fl-find-time 0))
  (let ((op (fl-eval body-part)))
    (if (null op) (return-from fl-find-time 0))
    (let ((time (time-required op)))
      (cond (time (fl-eval (first time)))
            ((body op) (apply #'add (mapcar #'fl-find-time (body op))))
            (t 0)))))
```

Figure 10.1: Implementation of the time-required checker.

```
Warning: Time-required in FLOWS::LPCVD-SILICON-NITRIDE [2 hours 45 minutes]
is less than the time-required in its body [3 hours]

Warning: Time-required in FLOWS::NITRIDE-WET-ETCH [1 hour 20 minutes]
is less than the time-required in its body [1 hour 35 minutes]

Warning: Time-required in (FLOW (:TIME-REQUIRED (:MINUTES 20))
                                (FLOWS::OXIDE-7-1BOE-ETCH :TIME
                                    (:MINUTES 11 :SECONDS 0))) [20 minutes]
is less than the time-required in its body [22 minutes]

Warning: Time-required in FLOWS::LPCVD-SILICON-NITRIDE [2 hours 45 minutes]
is less than the time-required in its body [3 hours]

Warning: Time-required in FLOWS::P-FIELD-PATTERN [2 hours 20 minutes]
is less than the time-required in its body [2 hours 50 minutes]

Warning: Time-required in FLOWS::NITRIDE-WET-ETCH [1 hour 20 minutes]
is less than the time-required in its body [1 hour 35 minutes]

Warning: Time-required in FLOWS::POLYSILICON-PATTERN [3 hours 35 minutes]
is less than the time-required in its body [3 hours 55 minutes]

Warning: Time-required in FLOWS::BACKSIDE-BPSG-WET-ETCH [30 minutes]
is less than the time-required in its body [30 minutes 20 seconds]

Warning: Time-required in FLOWS::BACKSIDE-POLY-PLASMA-ETCH [1 hour]
is less than the time-required in its body [1 hour 15 minutes]

Warning: Time-required in FLOWS::CONTACT-DESCUM-SET [25 minutes]
is less than the time-required in its body [1 hour 47 minutes]

Warning: Time-required in FLOWS::METAL-DEPOSITION [3 hours 15 minutes]
is less than the time-required in its body [4 hours 30 minutes]
```

Figure 10.2: Warnings issued by the time-required design rule check run on the CMOS baseline process.

Manufacturing policy checks such as this might be appropriate during any stage of design, but are especially important before transfer to a fabrication facility. Within the MIT Integrated Circuits Laboratory, for example, there are a set of rules specifying the allowable ranges of parameters for use within the facility (*e.g.*, high temperature furnaces may be operated only between 800 C and 1000 C and between 30 minutes and 4 hours). Process steps that are outside of these bounds require special approval (possibly indicating the need for an `:approval` operation attribute in the PFR). Many such rules can be captured implicitly in a library of operations provided by a facility. Incoming processes might then be restricted to only use these provided (manually verified and approved) operations. The direct and general specification of design rules remains an important need, because new operations would be continuously created within a facility and requested from outside the facility. Again, the representation of such rules should be made formal and susceptible for use within automatic design-rule checkers. Whether these should be a part of the Process Flow Representation (*e.g.*, as preconditions for the execution of operations), or expressed via an adjunct representation remains an open question.

10.3 Process Implementation Verification

The checking of a process description for conformance to some set of rules before using that description is only one form of process verification. Another notion of verification arises when one considers different aspects or attributes of a process description as *specification* of goals or requirements. Other descriptions or results of several kinds might then be compared against these requirements to *verify* satisfaction.

An especially important type of process verification arises if one considers some attributes or views within a process description to be *implementations* of other views. In the PFR, for example, mechanisms are provided for expressing changes in wafer state, treatments, and machine settings. The PFR itself makes no guarantees about

the consistency of these descriptions. Indeed, the PFR does not even mandate that one view is a specification and the others implementations: for example, the `:change-wafer-state` might be a characterization of the effect of a specified `:treatment`, or the `:treatment` might be an implementation of a specified `:change-wafer-state`. Instead, it is the role of external tools or users of the PFR to impose such meaning on the PFR, and to further verify consistency between views. Currently, this kind of verification is by no means automatic, and tools to help the user perform such verification are generally limited to process simulation programs.

Another type of process verification arises when one considers a process as a whole to be an implementation of *externally* imposed requirements. Specifically, there is a need to verify that the process will (via simulation) or does (via fabrication) produce wafer or device structures with specified geometric, mechanical, and electric properties. Verification of processes at this level is presently a long and people-intensive procedure, and is intimately woven into the larger process design problem. Automation of such verification will require a great deal of further research; the formal representation of device requirements would be a partial step in this direction.

10.4 Conclusions

Several kinds of *process verification* are possible and desirable to support fabrication process design, transfer, and manufacturing. Simple examples of these design rule checkers provide proof of the concept, though much work remains to define the most appropriate mechanisms for the representation of design and manufacturing rules and requirements, and for the automation of verification utilities.

Chapter 11

Process Traveler Generation

An important goal of the Process Flow Representation (PFR) is to better integrate design and manufacturing. In current practice, the descriptions of the process used for design and for manufacturing are often completely separate, and transfer of processes for fabrication requires rewriting in a form that the manufacturing system or staff understands or requires. In this chapter, an automatic conversion from PFR descriptions to a pre-existing manufacturing format is described.

11.1 Travelers and Opsets

A computer-accessible (but not manipulable) system of *travelers* and *opsets* has evolved within the MIT Microsystems Technology Laboratories (MTL). Originally designed by Wayne Frank of the MTL, this system provides a necessary minimal capability to specify, track, and record information about processing for wafer lots. An *opset file* is a short description (about one written page) of a collection of operations that together form a cohesive step within an overall process. A directory of standard opsets has evolved over time, including those for lithography, diffusion/oxidation, implantation, etching, and deposition. An oxidation opset, for example, usually contains a clean step, a furnace step, and a measurement or inspection step. A *traveler file*

specifies a sequence of opsets (by the name of the opset file) to be performed on a lot. During fabrication, the traveler file is consulted for the next opset to be performed, and updated whenever an opset is completed. Generally a copy of the standard opset file is also appended to the end of the traveler, and specified measurements and comments added to that copy of the opset file.

The traveler and opset has proven minimally sufficient to support the management of the MTL, despite the many limitations of the approach. Most of the limitations stem from an inability to manipulate the specifications of process parameters or the data captured in opsets. Furthermore, opsets tend to be minimal descriptions of steps sufficient for fabrication use only. For example, furnace steps specify recipe numbers, but do not carry information about the corresponding treatment, so that simulation based on opset information along is not possible.

11.2 Motivation for Traveler/Opset Generators

A key purpose of the CAFE system (Chapter 3) is to better support IC fabrication. Mechanisms based directly on the PFR are being developed to aid in process specification, operation scheduling, lot tracking, and data collection [15]. When fully developed, all information about a process will be integrated, accessible, and manipulable by the computer, so that design and manufacturing activities will have full access to both the data and tools used in each domain.

Mechanisms to generate traveler and opset descriptions remain important, however. First, an opset and traveler generator is needed in the interim before a direct fabrication interface for the PFR is available. Second, the opset and traveler generator are helpful during a transition period, when existing travelers and opsets are converted to PFR descriptions. By comparing an opset or traveler generated from a newly converted PFR description to the original opset or traveler, many errors in the PFR description can be detected. Finally, the generators are useful demonstrations

of the ability to interface to existing manufacturing mechanisms and systems. The opset/traveler system is typical of that in use in some facilities, while other manufacturing sites use commercial lot tracking systems. Conversion of PFR descriptions to the Workstream (COMETS) system [36], for example, should be possible.

11.3 Generator Approach

The approach taken in this work is to extract information solely from the PFR to generate opsets and travelers. Travelers and opsets have a more limited notion of process structure than does the PFR. A three-level hierarchy is used: a traveler is a sequence of (usually 50-100) opsets, which is a sequence of (usually 1-5) *primitive steps*. The main problem in converting PFR descriptions to opsets and travelers, then, is reducing the arbitrary hierarchy available in the PFR to this three-level hierarchy. In the approach taken here, additional conventions to PFR usage must be followed (particularly with respect to operation naming) to indicate this hierarchy.

Two separate generators have been implemented: one for writing a traveler file, and the other for writing opset files. In order to use either generator, one must first write the fabrication process using the Process Flow Representation as described in the PFR User's Guide (Appendix A). An operation or flow is identified as an opset using the `:opset` attribute, where a string giving the name of the opset is given as the value.

For each opset found in a specified flow, an entry is added to the resulting traveler. The traveler generator provides one additional degree of flexibility. Each entry on a traveler consists of three parts: a *descriptive name* for the step, the name of the opset itself, and an area for recording the date the opset is started and completed and the number of wafers used in the opset. Any definition (or function) name that evaluates to a *single* opset qualifies as a descriptive name for that opset. The first one of these that is encountered for each opset will be the descriptive name used in the traveler.

11.4 Traveler Generation Example

The traveler generation example draws on the MIT CMOS baseline process to show the typical use of the PFR and traveler generator. The top-level PFR description of the CMOS baseline process is shown in Figure 11.1. The first operation in the body,

```
(define cmos-baseline
  (flow
   (:doc "CMOS Baseline Process")
   (:body
    initial-epi  ;Start with Epi wafer

    well-formation
    active-area-definition
    field-formation
    channel-formation
    source-drain-definition
    bpsg-passivation
    contact-definition
    metal-definition)))

(define well-formation
  (flow
   (:body n-well-formation
          p-well-formation)))

(define n-well-formation
  (flow
   (:body stress-relief-oxide
          lpcvd-silicon-nitride
          n-well-pattern
          nitride-plasma-etch
          n-well-ion-implant
          resist-ash)))

;; Bindings to standard opsets...
(define stress-relief-oxide dsro430-set)
(define lpcvd-silicon-nitride dnit1-5k-set)
(define n-well-pattern phnwell-set)
(define n-well-ion-implant inwellpkt-set)
(define resist-ash ash-set)
```

Figure 11.1: Fragments of the PFR description of the CMOS baseline process.

`initial-epi`, is a PFR description of the starting material and is not defined to be

an opset. The second operation in the body, `well-formation`, has a short definition consisting of two sub-operations, `n-well-formation` and `p-well-formation` (the CMOS baseline process uses twin wells). The `n-well-formation` itself consists of six operations each of which corresponds to some opset from the library of standard baseline opsets (the definitions at the bottom of Figure 11.1).

This CMOS baseline PFR illustrates the typical use of opsets. Some hierarchical description of the basic structure of the baseline process finally results in calls to non-parameterized opsets. Each opset usually consists of several smaller operations and calls on utility functions that are made more flexible and general via parameterization. While the baseline process in Figure 11.1 shows that the PFR can closely mimic (or add hierarchical descriptions to) the old traveler-opset descriptions, it should be noted that a more complete migration away from the opset-traveler methodology is needed to take full advantage of the power and flexibility of the PFR.

The first few lines of the traveler that results from the cmos-baseline PFR description are shown in Figure 11.2. In the traveler, the higher-level structure of the baseline process has been flattened into a sequential listing of the opsets appearing in the baseline process.

11.5 Traveler Generator Implementation

The implementation of the traveler generator is relatively simple. The code for the generator, shown in Figures 11.3 and 11.4, consists of three basic functions. The first function is `make-traveler`. This sets up the core evaluator by creating a simple description of the wafer, and sets the opset counter to zero. It then calls the core evaluator on the form passed in (*e.g.*, the symbol `'cmos-baseline`), and calls the `traveler-interp` function on each member of the body of the resulting operation.

The second function, `traveler-interp`, is the core of the traveler generator. This does a recursive-descent into each part of the body, looking for uses of definitions

```
1   STRESS-RELIEF-OXIDE        Number wafers ________
    dsro430-set                Opset start   ________
                               Opset finish  ________

2   LPCVD-SILICON-NITRIDE      Number wafers ________
    dnit1-5k-set               Opset start   ________
                               Opset finish  ________

3   N-WELL-PATTERN             Number wafers ________
    phnwell-set                Opset start   ________
                               Opset finish  ________

4   NITRIDE-PLASMA-ETCH        Number wafers ________
    plnit1-5k-set              Opset start   ________
                               Opset finish  ________

5   N-WELL-ION-IMPLANT         Number wafers ________
    inwellpkt-set              Opset start   ________
                               Opset finish  ________

6   RESIST-ASH                 Number wafers ________
    ash-set                    Opset start   ________

...
```

Figure 11.2: The beginning of the generated CMOS baseline process traveler. corresponding to the PFR description of Figure 11.1.

that qualify as opsets (for which `opset?` returns a non-null opset name). If the form is an opset, then an entry is made on the traveler, and the interior of the opset is not examined further. If the form is not an opset, but is an operation, flow, or sequence, then `traveler-interp` digs further into the body looking for opsets. The third function in the generator is `generate-traveler-line`, which outputs a single entry on the traveler.

The traveler generator interacts with the PFR description partially at a textual level. That is, it must explicitly interact with the *names* of functions or definitions that are being used, and must do so before they are squeezed out by the core evaluator. This is a common feature of PFR interpreters: a textual PFR fragment is sometimes examined directly, and is sometimes evaluated to substitute the values of definitions

```
;; A Traveler Interpreter
;;
(defun write-traveler (flow file)
  (setf *traveler-stream*
        (open file :direction :output
              :if-exists :supersede))
  (make-traveler flow)
  (close *traveler-stream*)
  (setf *traveler-stream* t)
  file)

(defun make-traveler (form)
  (declare (special wafer *opset-number*))
  (setf wafer (make-simwafer))
  (setf *opset-number* 0)
  (let ((op (fl-eval form)))
    (cond ((operation-type? op)
           (dolist (body-part (body op))
             (traveler-interp body-part))))))

(defun traveler-interp (form)
  (declare (special *opset-number*))
  (let ((op (fl-eval form)))
    (cond ((opset? op)
           (generate-traveler-line
            (setf *opset-number* (+ 1 *opset-number*))
            form
            (opset? form)))
          (t
           ;;dig down looking for opsets...
           (cond ((operation-type? op)
                  (dolist (body-part (body op))
                    (traveler-interp body-part)))
                 ((fl-sequence? form)
                  (dolist (seq-part (fl-sequence? form))
                    (traveler-interp seq-part)))))))))
```

Figure 11.3: Code for the traveler interpreter.

```
(defun generate-traveler-line (opset-number call-name opset-name)
  (format *traveler-stream* "~3D   ~47ANumber wafers ________~%"
          opset-number
          (if (symbolp call-name)
            call-name
            (format nil "~S" call-name)))
  (format *traveler-stream* "      ~(~47A~)Opset start   ________~%"
          opset-name)
  (format *traveler-stream* "~53AOpset finish  ________~%~%" " "))

(defvar *traveler-stream* t)

;; Something is an ''opset'' if it is an operation or flow (which
;; can be interpreted or not), and if it has a non-null :opset
;; value (the first item of which is returned).
;;
(defun opset? (form)
  (let ((op (fl-eval form)))
    (if (operation-type? op)
      (first (operation-opset op)))))
```

Figure 11.4: Code for the traveler interpreter.

or to apply functions (in essence, to remove many of the language-like aspects of the description) and the resulting object is queried for more information. The traveler generator also illustrates the inherently recursive nature of the PFR and code that must manipulate the PFR.

11.6 PFR Opset Style

As in the traveler generator, an opset is identified as any operation or flow with a specified :opset attribute. Within the opset operation or flow, any valid PFR description may be used. The rules summarized in this section govern how the information in the PFR operation is used by the opset generator, and how the organization of the operation impacts on the appearance of the opset. The example opset shown in Figure 11.5 illustrates these various groupings of information. The PFR descriptions that result in this opset are discussed in Section 11.7.

```
!DIFFUSION                    !dsro430-set
LOT # !                       DATE (YYMMDD) !

Operation      Parameter           DATA ENTRY          Time Required
---------      ---------           ----------          -------------
rca-clean                                              !2 hrs !0 min
rca-sc1
               Machine             !rca
               Recipe              !Sc1
               Time                !(Minutes 10)

rca-hfdip                                              !0 hrs !2 min
               Machine             !oxide
               Sink                !Oxide-Sink
               Tank                !2
               Acid                !50-1Boe
               Time                !(Seconds 60)

rca-sc2
               Machine             !rca
               Recipe              !Sc2
               Time                !(Minutes 15)


oxidation                                              !5 hrs !0 min
               Thickness           !(Angstroms (Mean 430 Range 20))
               Machine             !GATEOXTUBE
                                   = tubeA1
               Recipe              !210


inspect-thickness                                      !0 hrs !15 min
  INSTRUCTION:Center Wafer oxide THICKNESS (5 readings) Spec (not specified)
               Machine             !ellipsometer
               Film-Type           !(Flows::Use-Film-Type-For-Machine
                                    :Film-Type "Oxide" :Machine "Ellipsometer")
                                   = oxide

               Center Wafer:Top    !
               Center Wafer:Center !
               Center Wafer:Left   !
               Center Wafer:Right  !
               Center Wafer:Bottom !

       Total time required: !7 hrs !15 min

Revision &1.1
```

Figure 11.5: The opset generated for the `dsro430-set` operation.

11.6.1 Header

The opset header is generated based on the name of the opset definition. At the left of the top line is a description of the process *category* that the opset falls into. This category is determined by the first one or two characters in the name of the opset. Currently, these are "diffusion" (d), "wet etch" (w), "photolithography" (ph), "plasma-etch" (pl), "ash" (a), or "ion-implant" (i). The full name of the opset itself appears at the right of the top line of the opset. While the MTL and individual users often use additional conventions to encode information into the name of the opset (such as the thicknesses of deposited or etched materials), none of this information except the opset category is used by the opset generator.

A header line is also output to label the columns of information that will form the bulk of the opset. These columns are `Operation` to provide summary names or descriptions of the sub-operations making up the opset, `Parameter` to describe parameters of the sub-operation (these may be settings, readings, or other information as described further below), `Data Entry` to show the values of both PFR-specified parameters and user or operator recorded values, and finally `time-required` to show the length of time required to complete the sub-operation.

11.6.2 Instructions

Next, any instructions that are attached to the top-level opset flow are printed. These are prefaced with the word `INSTRUCTION:`, followed sequentially by all of the instructions that appears (at the top level only) of the opset. Instructions that are contained within other sub-operations of the opset are printed when the body of the opset is examined later.

11.6.3 Machine

If any machine is attached to the top-level opset, it will appear. Under the parameter column the keyword `Machine` will be shown, and the value of the machine slot from the operation displayed under the data entry column. If the machine is a symbol (such as `GateOxTube`) or some other form (such as a function call that calculates the machine name), the symbol or form will also be completely evaluated, and that value shown following an equal sign (=) in the data entry column.

11.6.4 Settings

If machine settings for the operation are available, each of the keyword-value pairs will be shown. Each keyword will appear as a parameter of the opset, and the value of that keyword shown in the data entry column. Again, if the value supplied in the PFR differs from the evaluated version of that value, the evaluated form will be shown following an = in the data entry column.

11.6.5 Body

The generation of information for the opset is most complicated by the fact that an opset is usually a flow or an operation that consists of more than one smaller sub-operations. For each of the members of the body of the operation, the following procedure is followed.

First, some descriptive name for the sub-operation is generated. If the sub-operation is an invocation of another definition, then the name of that definition appears under the `operation` column. If a function rather than a simple definition is invoked, then the name of the function is used, and a summary of the function arguments is printed under the parameter and data-entry columns. If an anonymous or in-line operation or flow is used, then there is no definition or function name available to describe the sub-operation. In this case, the change-wafer-state view is consulted,

and a reasonable name (as well as parameter and data-entry columns summarizing the data in the change-wafer-state view) is constructed from that view. Note that for this view, if more than one CWS primitive appears, then `operation`, `parameter`, and `data-entry` columns are printed for each and every CWS primitive in the view, with the addition of the word `simultaneously` beside them so that it is clear that the sub-operation is having more than one effect on the wafer at the same time. If a change-wafer-state view is not present for the sub-operation, then treatment information is next consulted. Multiple treatment primitives are interpreted as occurring sequentially rather than simultaneously. In addition to the descriptive name for the sub-operation, the time-required for the sub-operation is calculated and printed in the `time-required` column.

Once a descriptive name for the sub-operation has been output, instructions, machine, and settings information is again generated as for the top-level operation. Similarly, if the sub-operation consists of still further sub-operations, the above procedure will be repeated for each of these smaller operations. Once the body of the sub-operation has been processed (recursively), then readings lines as described below are generated.

11.6.6 Readings

If readings have been specified in the opset definition (or within any sub-operations) then additional lines are added to the opset output. For each reading entry, the string descriptor of the reading provided in the PFR is output in the `parameter` column, and space is left in the data-entry for the operator to fill in the value of that parameter based on measurements or other information.

11.6.7 Time-Required

If a time-required field has been specified for the top-level, then a line showing the `Total Time Required` to perform the entire opset is printed. Note that there is no

guarantee that this time will equal the sum of the times required in the sub-operation, or even that this total time will be less than the sum of the time-required in the sub operations. That is to say, the PFR and the opset generator do not automatically check for time-required consistency; one must run a separate design-rule checker over the process flow to look for these or other inconsistencies.

11.6.8 Version

Finally, the `version` view of the top-level opset is consulted to try to locate the last version number attached to the opset. If one is found, that number is output, otherwise the version number `0.0` is output.

11.7 Opset Generation Example

The following example illustrates the use of the opset generator. This example uses the PFR description of the stress-relief-oxide opset, `dsro430-set`, as shown in Figure 11.6. The resulting opset is shown in Figure 11.5.

The PFR description of an operation generally contains more information than what has been kept in or will be output as the opset. For instance, the documentation and version slots for the `dsro430-set` operation are more thorough and complete than in the resulting opset file. Furthermore, one finds that the hand-generated opsets are usually restricted to machine, settings, and readings-like information. The change in wafer state information is usually omitted almost entirely, though occasionally such information may appear as specifications during measurements, and may also be summarized as part of the coded opset name. Treatment information is rarely kept in the old opsets. This lack of information in opsets makes writing Suprem-III or other process simulation input files extremely painful, and makes transfer of the process difficult.

Still further information is kept in the PFR operation that does not appear in

```
(define dsro430-set
  (flow
   (:doc "Stress Relief Oxide.  Purpose is to minimize
    the stress effects of nitride deposition and
    processing. Operation consists of a clean, a furnace
    step, and an inspection.")
   (:version
    (:modified :number 1.0 :by "Duane Boning"
               :date "February 27, 1989"
               :what "Write operation in flow language")
    (:modified :number 1.1 :by "Duane Boning"
               :date "April 7, 1990"
               :what "Conform to opsets."))
   (:time-required  (:hours 7 :minutes 15))
   (:body
    (flow
     (:doc "These two steps have to be done right
      after each other")
     (:permissible-delay :minimal)
     (:body
      rca-clean
      (flow
       (:doc "SRO furnace processing")
       (:change-wafer-state
        (:oxidation
         :thickness (:angstroms (:mean 430 :range 20))))
       (:treatment
        (furnace-rampup-treatment
         :final-temperature (:mean 950 :range 10))
        (furnace-dryox-treatment
         :temperature 950 :time (:minutes 100))
        (furnace-rampdown-treatment
         :start-temperature 950))
       (:time-required (:hours 5 :minutes 0))
       (:machine GateOxTube)
       (:settings :recipe 210))))
    (inspect-thickness :where "Center Wafer"
                       :film-type "oxide"
                       :machine "ellipsometer"))))
```

Figure 11.6: The PFR definition of the `dsro430-set` opset.

the opset. The `dsro430-set` flow consists of two sub-operations: an in-line definition of a flow, and the `inspect-thickness` function invocation. The first flow is simply a grouping together of two smaller operations: an `rca-clean`, and another in-line operation. For this first flow, there is a `(:permissible-delay :minimal)` form, indicating that there should be minimal delay between the end of the `rca-clean` and the furnace processing flow. This permissible delay is currently not shown in the output opset.

An appreciable fraction of the `dsro430-set` information is, however, shown in the resulting opset. First, one sees that the individual steps in the rca-clean have been generated. The PFR description of the `rca-clean` operation is shown in Figure 11.7. The `rca-clean`, as well as the operations it calls, belong to a set of utility definitions and functions. Most opsets tend to use these basic utility operations. Using the PFR, these can be encapsulated and shared, while duplication (and often inconsistencies) are required when writing opset files by hand.

If one looks at the generated opset file of Figure 11.5, one sees that following the rca clean steps an oxidation operation occurs. In the `dsro430-set` operation, this corresponds to the in-line definition of a furnace processing flow. In this case, the name `oxidation` has been generated using the change in wafer state information, and the thickness that is intended has also been shown. In this case, the treatment has not been examined at all. The oxidation information is followed by the machine and the recipe number for the oxidation.

The final step in the body of `dsro430-set` is the `inspect-thickness` function call. The implementations of this (and of the utility functions it uses) are shown in Figure 11.8. An instruction line is added to the opset shown in Figure 11.5 for the inspection operation name and instructions, followed by lines for the machine name and settings. Several lines are then added to generate data entry slots for the various readings that are requested in the `:readings` view of the `inspect-thickness` operation.

```
(define rca-clean
  (operation
   "General purpose RCA clean operation,
    with short 50-1 HF dip."
   (:time-required (:hours 2))
   rca-sc1      ;Organic Clean
   rca-hfdip    ;Oxide Clean
   rca-sc2      ;Ionic Clean
   ))

(define rca-sc1
  (operation
   "Organic Clean"
   (:machine "rca")
   (:settings  :recipe "sc1"
               :time (:minutes 10))))

(define rca-hfdip
  (oxide-50-1boe-etch :time (:seconds 60)))

(define rca-sc2
  (operation
   "Ionic Clean"
   (:machine "rca")
   (:settings  :recipe "sc2"
               :time (:minutes 15))))

(define (oxide-50-1boe-etch time)
  (oxide-boe-etch :acid :50-1boe :time time))

(define (oxide-boe-etch time acid)
  (operation
   "Generic BOE etch operation.
    Parameters include the buffered-oxide
    etchant mixture and the etch time."
   (:change-wafer-state
    (:etch :material "oxide"
           :thickness (* time (etch-rate
                               :acid acid))))
   (:machine "oxide")
   (:settings :sink Oxide-Sink :tank 2
              :acid acid :time time)
   (:time-required (* time 2))))
```

Figure 11.7: The PFR definition of rca-clean operations used by the `dsro430-set` opset.

```
(define (inspect-thickness
         film-type (spec "(not specified)")
         (where "") where2
         machine (time (:minutes 15)))
  (operation
   (:time-required time)
   (:machine machine)
   (:settings :film-type
              (use-film-type-for-machine
               :film-type film-type
               :machine machine))
   (:readings (taking-5-readings
               :what :thickness :where where
               :where2 where2))
   (:instructions
     (inspect-instructions-for-5-readings
       :what :thickness :film-type film-type
       :where where :where2 where2
       :spec spec))))
```

Figure 11.8: The PFR definition of the inspection function used by the `dsro430-set` opset.

11.8 Opset Generator Implementation

The implementation of the opset generator is more complicated than that of the traveler generator, and consists of about 400 lines of CommonLisp code (in addition to the core evaluator). The essential aspects of the opset generator, however, are similar to the traveler generator, but more elaborate extraction of information from the PFR and output formatting are required [87].

11.9 Discussion: PFR Modifications

An earlier implementation of the opset and traveler generators [88, 87] avoided the addition of the :opset attribute to the PFR, and instead depended on naming and style conventions to indicate which operations were opsets. These additional conventions proved confusing and cumbersome when writing PFR code. The :opset attribute, on the other hand, enables one to clearly and cleanly identify operations

```
(define (inspect-instructions-for-5-readings
         what where where2 spec film-type)
   (if where2
       (|| where " and " where2 " " film-type
        " " what " (5 readings each) Spec "
        (flow-string "~A" spec))
       (|| where " " film-type " " what
        " (5 readings) Spec "
        (flow-string "~A" spec))))

(define (taking-5-readings readings what
                           where where2)
   "The basic instructions and inquiries
    for making 5 readings around a wafer"
   (if where2
       (sequence
         (what (|| where ":top"))
         (what (|| where ":center"))
         (what (|| where ":left"))
         (what (|| where ":right"))
         (what (|| where ":bottom"))
         (what (|| where2 ":top"))
         (what (|| where2 ":center"))
         (what (|| where2 ":left"))
         (what (|| where2 ":right"))
         (what (|| where2 ":bottom")))
       (sequence
         (what (|| where ":top"))
         (what (|| where ":center"))
         (what (|| where ":left"))
         (what (|| where ":right"))
         (what (|| where ":bottom")))))
```

Figure 11.9: The PFR definition of utility functions used by the inspection operations.

as "opsets" for manipulation by programs that know what this means. An important lesson is that the PFR must be kept extensible in order to support new application programs.

11.10 Conclusions

A PFR-based traveler and opset generator has been implemented which converts PFR process flows into MTL standard opset and traveler descriptions. These descriptions can be used to check PFR descriptions of processes against older MIT MTL processes, and can also be used to ease the migration to the Process Flow Representation. By drawing on the core PFR evaluator, the implementation of the opset and traveler generators is straight-forward and illustrates basic aspects of the programmatic interface to the PFR. These generators demonstrate the feasibility of building interfaces between the PFR and an internal process traveler system. A connection to other existing and widely used process specification systems, such as COMETS or WORKSTREAM, would be interesting. While other research has demonstrated a connection from manufacturing to simulation [89], a PFR-based process description and interfaces to both simulation and manufacturing systems such as COMETS might result in better integration of design and manufacturing activities (as well as contribute to the acceptance of the PFR. Ultimately, a manufacturing system such as CAFE [6] is needed that can use the PFR more to facilitate and integrate process design and fabrication.

Part IV

Methodologies for Process Design

Chapter 12

Methodology Paths

A new way of looking at process design is needed in order to achieve the vision of application specific processes. It is first necessary to understand what is meant by a "methodology":

> A dictionary definition of the term *algorithm* is a step-by-step procedure for solving a problem. *Method* is defined as a systematic way, technique, or process of or for doing something; a body of skill and techniques. A body of methods, procedures, working concepts, rules and postulates employed by a science, art, or discipline is known as a *methodology*. Finally *strategy* is defined as the art of devising or employing plans toward a goal. [90]

This thesis has so far addressed some of the representations and tools that are needed to support the design of application specific processes (ASP). In this chapter, additional concepts, methods, and procedures that contribute toward ASP design are considered. The guiding idea behind the ASP design methodology is to shift away from traditional tool-centered design toward data-centered process design, so that process design becomes a series of transformations of design data. Several different categories of design data, and paths by which that data is generated or manipulated can be identified. These paths carry task-oriented names, including "implementation", "simulation", and "characterization" paths. Because technology CAD is still a

relatively young field, much work remains to be done in understanding, defining, and experimenting with possible design methodologies [91, 92, 93].

12.1 Design by Data Transformations

In Chapter 4, a generic model of semiconductor processing was discussed in which the state associated with wafer production was identified and partitioned, and models of the transformations of that data based on physical causality were described. The act of design is also a type of process, and a "generic design model" of sorts can be identified. First, the data that is needed to support the specification and design of a fabrication process must be identified. In addition to the categories of process information in the generic model (wafer state, changes in wafer state, treatments, machine settings, *etc.*), we should also add device performance (the electrical or mechanical characteristics of wafer structures). These will serve as the high-level "specification" for what a wafer structure and an associated process sequence must produce.

Once the types of design data have been identified, understanding of the possible and necessary transformations of that data is also needed. The transformation of one category of design data to another is achieved via some *task*. A graph of such tasks connecting the design data can be termed a *methodology* or *activity path*.

Implementation Path

The *implementation path*, shown in Figure 12.1, flows from the "high" level design data downward to more specific data. Each implementation task in the path (left to right arrows) uses some input design data as a *specification* and produces new design data that is the *implementation* of that specification.

Measurement Path

The *measurement path*, shown in Figure 12.2, flows from the right to the left. In general, we measure the response of some physical implementation to test conformance to a specification. For example, *device measurement* is the extraction of electrical characteristics from an actual wafer or device structure.

Simulation Path

The *simulation path* shown in Figure 12.3 is similar to the measurement path, except that representations of some implementation is tested rather than some actual object.

12.2 Discussion

Only a few of the tasks identified in these activity paths are well understood, and only a few of these are typically supported by solid design data representations and transformational tools. Those that are supported are primarily the areas of process simulation (transforming treatment to wafer structure information), and device simulation (transforming wafer structure data to device performance data). The data transformations in the reverse direction have traditionally been ill-defined and poorly supported or automated. There of these design data transformations (device design, structure implementation, and treatment synthesis) are discussed further below.

Device design is now carried out in an entirely ad-hoc fashion. The first step toward a device synthesis capability (and the automation of device design) is to develop solid representations for specifying device performance and wafer structures. This part of device design seems susceptible to future research. Beyond representations, a great deal of additional understanding of how device design is now carried out will be required before there is much hope in automating or aiding in device design. In the case of relatively well understood devices (MOS structures, for instance), some

work has been done in terms of device diagnosis that might also be applicable to device synthesis [86].

Once a final device or wafer structure has been postulated that achieves some desired device performance, the next necessary data transformation is the generation of a sequence of wafer *mutations* (or changes in the wafer state) that can transform some canonical starting physical wafer into the postulated wafer structure. Simple methods to help in this step (also known as *process integration*) are discussed in Chapter 13.

Once a mutator sequence has been found, the next necessary data transformation is the generation of treatment information for most if not all of the mutators (there may be some that can be transformed directly to machine settings or recipes). The *Process Advisors* discussed in Chapter 9 are examples of tools to aid in process synthesis.

12.3 Conclusion

An analogy between semiconductor processes and design processes exists. Just as we wish to automate fabrication processes, it is desirable to move toward the automation of process design. Additional research remains before such automation is possible. The approach used in this thesis to understand the state and transformations of state information (the generic process model) offers an interesting avenue for future research into understanding the design data and data transformations (design activities) involved in process design.

Incremental steps contributed by this thesis toward fabrication process design automation include the introduction of tools to aid in process simulation (the *Simulation Manager*), to aid in treatment synthesis (the *Process Advisors*), and new abstractions and methods for structure implementation (the *mutators* proposed in Chapter 13)

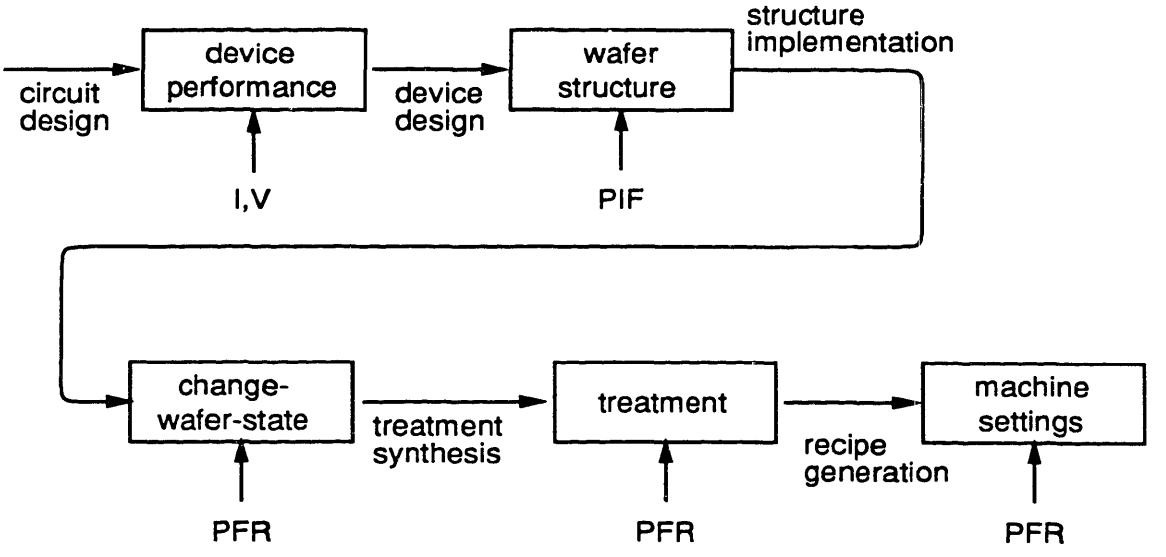


Figure 12.1: Implementation path.

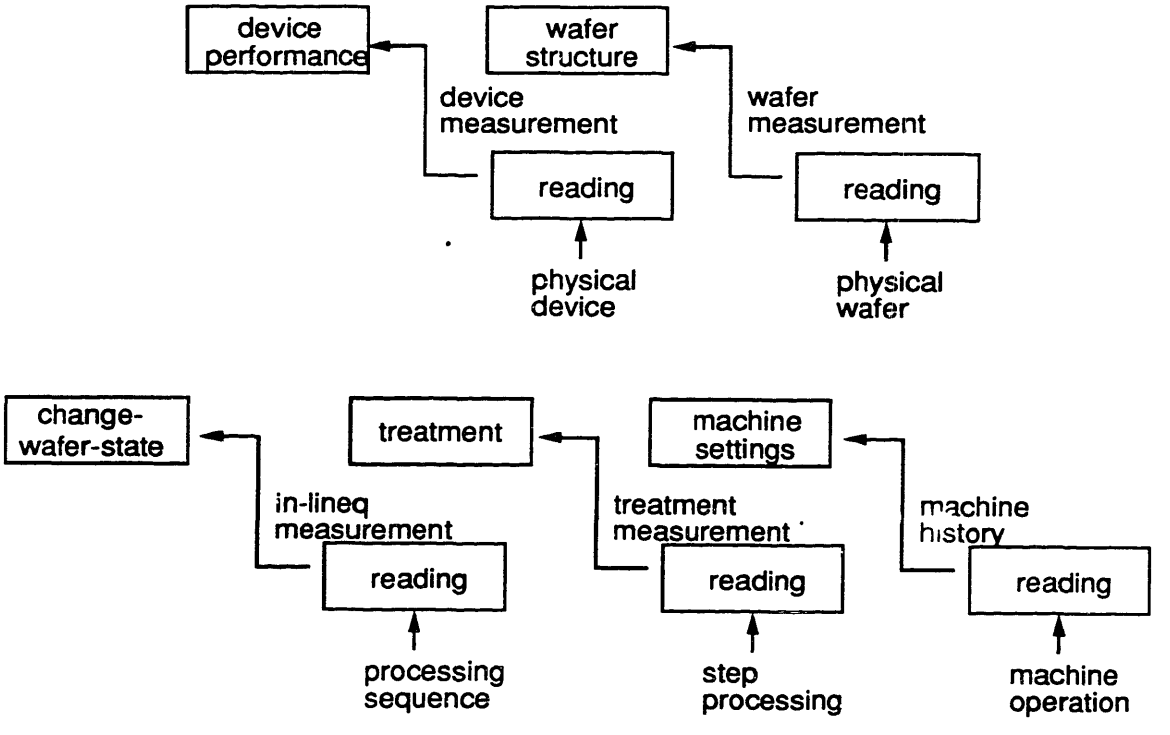


Figure 12.2: Measurement path.

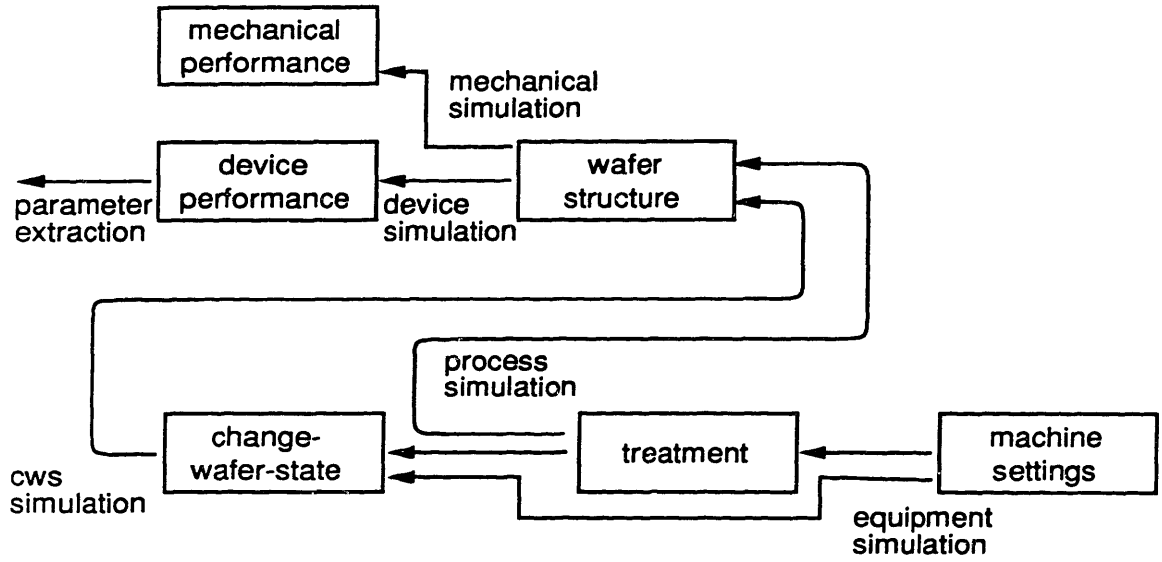


Figure 12.3: Simulation path.

Chapter 13

Process Integration: Mutators

The central argument of this chapter is that a way of making the conceptual idea of "change in wafer state" more *concrete* is needed. The problem is that the very term "change in wafer state" shifts focus away from the most important aspect of the idea: it is the *change* that is crucial, and not the *wafer state*. Referring to abstractions of unit operations that are performed on wafer states as "changes" is like referring to transistors as "changes in electric signals", or referring to gates as "changes in logical signals".

In this chapter, an analogy between logic design and process design is considered. First, the *wafer state* (as defined in Section 4.4.1) is identified as analogous to logical signals. Second, a *mutator* is defined to be an abstraction of the change in wafer state, and mutators are made analogous to logical gates.

A simple logic circuit schematic can be thought of as a directed monopartite graph of logic gates *nodes*, where *edges* indicate the wiring between gates. If one also defines the input and clock signals, it is possible to label all of the other edges of the graph with the resulting logic values. In this case, the schematic can be described as a directed bipartite graph, with nodes of both logic gate and logic value types, and edges serving as the physical connections between gates.

In a similar fashion, the sequence of processing steps can be pictured as a bipartite

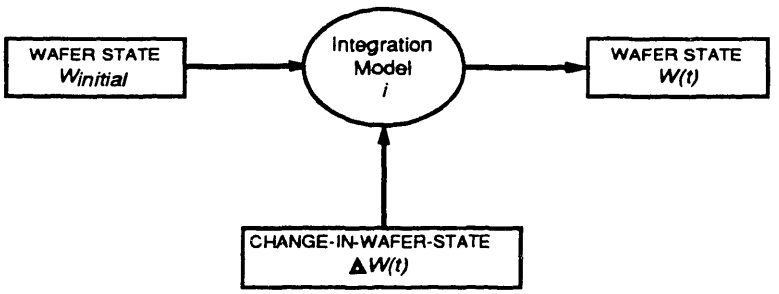


Figure 13.1: A graph of the wafer state and change in wafer state as suggested in the generic process step model.

graph, with *wafer state* and *change in wafer state* as the nodes, and time sequencing as the edges. Recalling the generic process model discussion of Chapter 4, these were pictured as in Figure 13.1.

Conceptual logic design is aided by three mechanisms:

- Abstraction of collections of transistors into "gates"
- A pictorial or schematic representation of gates
- Combination rules about connecting gates together (both as to how they may legitimately be connected, and the results of those connections).

The same steps for process design are possible, defining:

- abstract collections of unit process steps are abstracted together as "mutators"
- a schematic representation of mutators is proposed
- combination rules about connecting mutators are proposed, including
  - how and when mutators can and cannot be connected
  - the relationship between input and output signals when mutators are connected.

13.1 Proposed Mutators

A small set of mutators are here proposed. These mutators have corresponding pictorial or iconic representations, as shown in Figure 13.2. Mutators bear close resem-

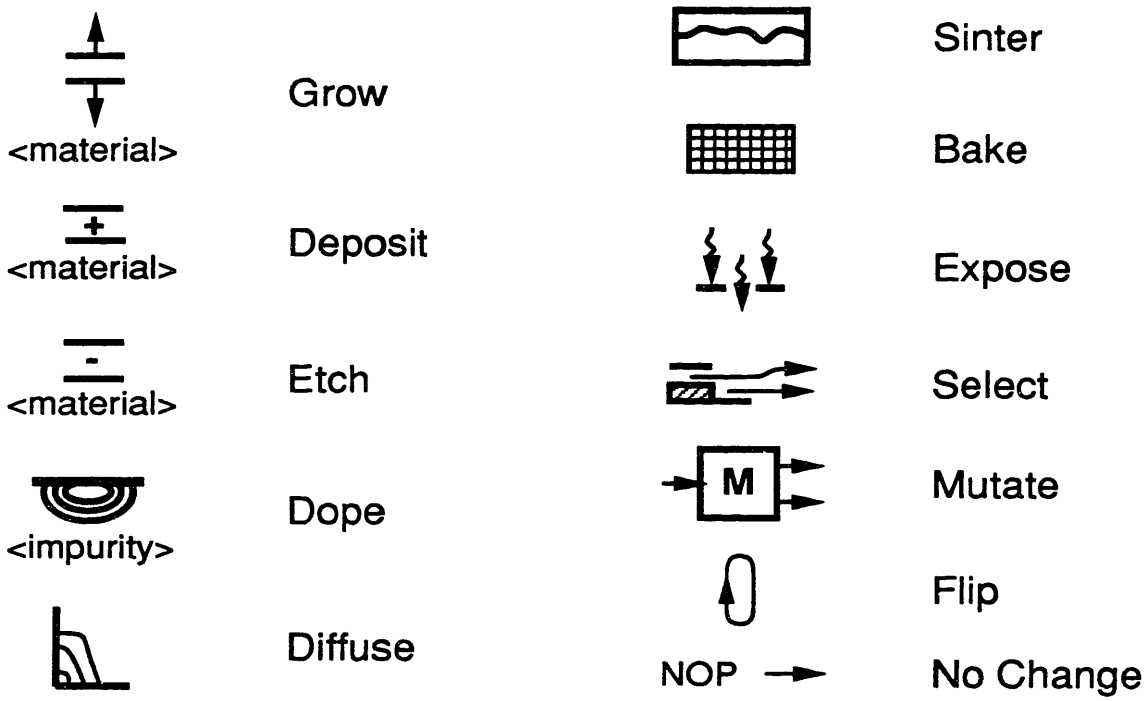


Figure 13.2: A set of mutator symbols.

blance to object oriented process representation approaches. In effect, the mutator is a logical extension of calling a process step an "object" – where a "process object" takes on a concrete picture and meaning in the form of an iconic mutator. Each mutator can be identified with a corresponding change in wafer state description, and may have combination rules associated with it. These mutators are neither exhaustive nor complete; they do, however, form a minimal working set for the exploration of the mutator concept.

The *grow* mutator involves the conversion of underlying material layers during the addition of a new layer. The PFR `:oxidation` change in wafer state primitive is a specialized example of the general *grow* mutator. The grow mutator combination rules depend on the material being grown. For example, an oxide or nitride *grow* mutator cannot be used if the wafer has metal already on it. The *deposit* material mutator differs from the *grow* mutator in that conversion of underlying material layers in general does not occur. The *deposit* mutator directly corresponds to the `:deposit` PFR change in wafer state primitive. The *etch* mutator removes material from the wafer. The introduction of dopant is caused by the *dope* mutator, and the *diffuse* mutator may cause the relocation of dopants within the wafer. The *selection* mutator is different from other mutators in an important way. The *selection* causes different mutations to occur at different locations on the wafer.† Common examples of selection are the exposure of light through a mask, the blocking of ion implantation by photoresist, or the selective growth of oxides or epitaxial silicon. The typical photolithography operation consists of several primitive mutators. In addition to *deposit* (for spinning resist), *selection*, and *etch* (for developing resist), the *expose* mutator causes the exposure of the wafer to light, and the *bake* mutator causes the cross-linking or breaking of bonds in a photoresist. The remaining primitive mutators include *sinter*, *flip*, and the *null* mutator. The *flip* mutator causes the wafer to be

†The recognition that other mechanisms beyond photolithography are also selectors is due to Robert Harris.

flipped over so that one-sided processing will occur on the back of the wafer rather than on the front. The *sinter* mutator changes the nature of the interface between metal or metal compounds and silicon. Finally, the *null* mutator has no effect on the wafer. In addition to the primitive mutators already described, the *block* mutator is the encapsulation of some number of other more detailed mutators into a single block. Just as a block of logic might be represented by a "truth table" or other logic description, the mutator block might be described via a table of input and output wafer states.

Just as only a limited set of logic gates or logic gate implementations are available in different technologies, so too might only a limited set of mutators be available at any one fabrication site. For example, some sites might not support the growth of epitaxial silicon. Similarly, the mutator combination rules may also differ from one fabrication line to another. For example, one facility may support one level of metal interconnect, while another supports two or three. In general, then, the implementation of a process depends on the basic set of mutators provided by a facility, as well as the "design" or combination rules of the facility. If a basic, minimal set of mutators and mutator combination rules were provided by every facility, and if processes were designed that only made use of this "standard" set, then such processes should, in principle, be completely portable.

13.2 Mutator Methodologies

Mutators may provide a mechanism to facilitate the *process integration* task. Recall that the object of this task is to produce a sequence of process steps that will produce the desired final wafer or device structure.

One-Dimensional Constructions

The above mutators are essentially one-dimensional in nature. To use them, a designer must define some number of one-dimensional cross sections of interest. The device structure is then simplified to a simple stacking of layers with certain properties. It is then possible to construct a simple process sequence that implements or produces the desired one dimensional structure. Goals of that sequence include *minimality* (the shortest possible sequence), and may include other considerations such as cost or flexibility.

Sequence Merge

Because processing takes place on complete wafers simultaneously (at least with most current technologies), the multiple one dimensional process sequences must be merged. Generally, this requires the insertion of *select* mutators to differentiate between cross sections, and the unification of process sequences so as to obey the general mutator combination rules. Rules involving the combination of select mutators will limit the possibilities. For example, a select mutator ranging over an ion implantation might involve photoresist as an implant mask; the same selector cannot select across diffusion (that is to say, diffusion is, in general, not a selectable mutator). Oxidation, on the other hand, is a selectable mutator (as it can be masked or prevented by nitride).

Sequence Manipulations

Other manipulations of the sequence might be possible, including reordering of the sequence, insertion of steps to facilitate merged processes, and elimination of steps to achieve minimal processes. Rules can be envisioned to guide each of these manipulations. These rules might be informal so as to guide manual manipulations, or formal so as to support automatic manipulation of the process.

The mutators presented so far are relatively limited. Not only are they one dimen-

sional in nature (excepting the *select* mutator), but they are also extremely limited in the flexibility that they provide. That is, they are only *parameterized* in a very limited way: some of the mutators provide material or impurity type parameterization. Additional parameterization is possible, based precisely on the parameters provided by the *change in wafer state* models and descriptions in the generic process model. In essence, the current mutators say *what* a process step does; a limited extension would be to parameterize mutators further to describe *how much* a process step does. For example, the grow, deposit, etch, and diffuse mutators might all be parameterized in terms of a thickness or distance. Additional combination rules and reasoning mechanisms would be required to support this additional parameterization.

13.3 Conclusion

Clearly, mutators are at a very early and primitive stage of development. They hold out the promise, however, of helping to

- understand and capture the kinds of mental processes and methods that a process expert uses to design a process (*i.e.*, consideration of one dimensional processes, forming a merged process, *etc.*).
- provide a graphical mechanism for sketching processes to help communicate basic process information to either other individuals or to a machine (via manual or computer-assisted drafting mechanisms)
- provide the basic objects for the future automated generation and manipulation of process sequences by intelligent process integration tools.

Chapter 14

Case Study: CMOS Baseline Process

This chapter describes the methodology used to perform simulations of the MIT CMOS baseline process. The process description for the baseline process is is described in Section 14.1. The use of the Simulation Manager and SIMPL-2 translator to produce models of the wafer via Suprem-III and SIMPL-2 process simulation is discussed in Section 14.2. Analyses of the wafer structure and impurity profile plots are summarized in Section 14.3. Finally, the generation of a two-dimensional profile from the one-dimensional process simulation results, and the use of Minimos to perform device simulations are presented in Section 14.4.

14.1 Baseline Process Description

14.1.1 PFR Flow

The first step in analyzing the CMOS baseline is the generation of a PFR description of the process. Information about the CMOS baseline process necessary to construct a PFR description came from several sources, including printed documentation [94], the many files containing "travelers" and "opsets" for the baseline, and consultation

Layer Name	Layout Function
CW	p-well
CD	diffusions (active areas)
CS	p+ select
CP	poly
CC	contact cuts
CM	metal1

Table 14.1: CMOS baseline process device layout layers.

with those responsible for the process [†]. The resulting baseline process requires about 1900 lines of textual PFR code, and is much too long to describe in detail here. Several fragments from the process are used as examples in Chapter 11. The basic structure of the baseline process is shown in Figure 14.1. Each leaf in this tree is identified with a baseline "opset".

14.1.2 Layout and Masking Information

An important part of the specification of a process is the definition and use of layout, mask, and cross section information. In order to perform one-dimensional simulations, for instance, one must know what process a particular cross section "sees" as a result of masking in that section. To facilitate this, the following definitions are made. The *layout layer* is a named layer as it appears on a layout CAD system, or as named in a CIF representation of a layout. In the MIT CMOS process, two different types of layout layers are used: first, the basic CMOS device layers, and second, a set of verniers for alignment and measurement of misalignment. These layers are transformed into physical *masks* as discussed in Chapter 7. In general, a mask

[†]The CMOS baseline process was developed by Prabha Tedrow, and the author thanks her for the many discussions about the process. Nestore Polce helped rework the baseline PFR to make its organization conform to the way that both designers and those fabricating the process think about it.

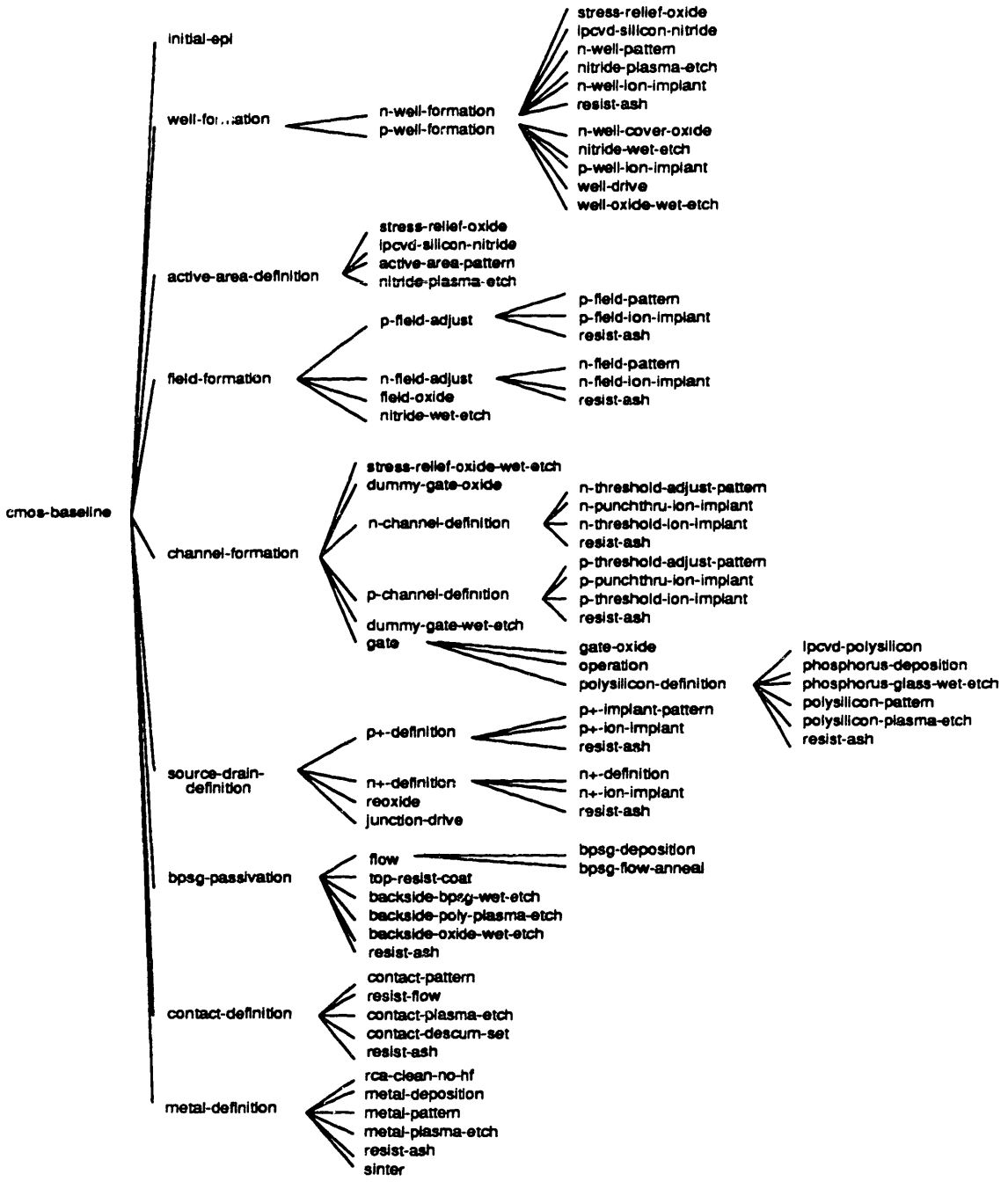


Figure 14.1: The high-level structure of the CMOS baseline Process in the PFR.

Layer Name	Layout Function
$CPWV$	p-well verniers
$CPFV$	p-field implant verniers
$CNFV$	n-field implant verniers
$CPTV$	pchan Vt implant verniers
$CNTV$	nchan Vt implant verniers
$CPPV$	p+ s/d verniers
$CNPV$	n+ s/d verniers

Table 14.2: CMOS baseline process vernier layout layers.

is generated through logical combinations of layout layers and through inversion of the mask (dark field masks). For instance, the nwell cover mask CNF is generated by merging the pwell (CW) and n-field implant vernier ($CNFV$) layers, and then converting to a dark field. This mask can be expressed in terms of the layout layers as the logical expression $\overline{CW + CNFV}$. The various device layers, vernier layers, and mask definitions used in the MIT CMOS baseline process are summarized in Tables 14.1, 14.2, and 14.3.

14.1.3 Cross Section Definition

In order to specify locations of interest for process simulation, one-dimensional *cross sections* (or "drill holes" on the layout) are defined as described in Chapter 7. For instance, the "nchan" cross section (through the channel of the nmos device in the pwell) is defined as the point on the layout where the pwell, active area, and poly layers overlap, or $nchan = CW*CD*CP$. Layer names not appearing in a section definition imply the inverse of that layer (one could have written $nchan = CW*CD*CP*\overline{CS}$). The six cross sections of interest for the one-dimensional simulation of the baseline process are summarized in Table 14.4.

Mask	Function	Layer Equivalent
CPW	Well Definition	$CW + CPWV$
CD	Active Area	CD
CPF	Cover N-Well	$\overline{(CW + CPFV)}$
CNF	Cover P-Well	$CW + CNFV$
CNT	Nchan VT Adjust	$\overline{(CW + CNTV)}$
CPT	Pchan VT Adjust	$CW + CPTV$
CP	Polysilicon	CP
CPP	P+ Implant	$\overline{(CS + CPPV)}$
CNP	N+ Implant	$CS + CNPV$
CC	Contact Cuts	$\overline{(CC)}$
CM	Metal Pattern	CM

Table 14.3: CMOS baseline process mask definitions.

Cross Section Name	Layer Definition
nchan	$CW * CD * CP$
ndrain	$CW * CD$
nfield	$CW * CP$
pchan	$CD * CP * CS$
pdrain	$CD * CS$
pfield	$CP * CS$

Table 14.4: CMOS baseline process cross section definitions.

14.2 Process Simulations

The Simulation Manager described in Chapter 7 takes the PFR description of the process and simulates on demand a specified cross section. The PFR process description includes information about the sense of the resist being used (positive or negative), and the name of masks for exposure operations. With this information and the definitions of cross sections, the translation to a "Suprem-III" [24] representation of the process can be made, accounting for the presence or absence of photoresist on subsequent processing steps (ion implantation and etching in particular). In addition to process translation, the Simulation Manager executes the Suprem-III simulator in an optimal way to generate Suprem-III "save structure" files containing descriptions of the wafer for later analysis. The sequence of screens and requests made to the Simulation Manager to perform the CMOS baseline simulations are summarized below:

1. Invoke the Simulation Manager. From the CAFE top level menu, the "Process Flow" menu, followed by the "Simulation Menu" and finally the Simulation Manager option are invoked.
2. Define flow screen responses:
   (a) Flow file:
   /caf1/a/boning/flow/baseline.v3/cmos-baseline.fl
   (b) Flow to simulate: cmos-baseline
   (c) Sections to simulation: (nchan pchan ndrain pdrain nfield pfield)
3. Define simulation environment responses:
   (a) Simulation Directory:
   /welles/a/boning/flow/sim/p-high-bg
   (b) Simulation Machine: welles
   (c) Giraphe Display: welles:0.0
4. Define wafer responses:
   (a) Material: silicon
   (b) Orientation <100>

(c) Impurity: boron

(d) Concentration: 1e19

(e) Make Wafer: push

5. Interpret Flow: push [and then wait about 5 minutes]

6. Simulation Matrix

(a) Reset Matrix Detail Depth: 3

(b) Simulate All: push [And wait about 2 hours for simulations to complete].

At this point the simulations have been completed, and individual steps can be examined using the Simulation Manager analysis screen. To generate reports based on the simulation results, it is often more convenient to return to the Unix shell on the machine where the simulations were run, and perform analyses (interactively or via scripts) directly on the resulting "save structure" files.

In addition to Suprem-III simulations, the SIMPL-2 translator (Chapter 8) was used to provide input for SIMPL-DIX [81]. The simulated final cross section for the baseline process is shown in Figure 14.2.

14.3 Analysis

In this section, a number of utilities are first described that have been written to make the chore of Suprem-III simulation and analysis more convenient from a UNIX shell, and for use by within the Simulation Manager. The commands and scripts written and used to generate analyses of the baseline process are then described.

14.3.1 Suprem-III Utilities

Several utilities facilitate the task of simulating the baseline process and help in the evaluation of the simulated results. These utilities are summarized below. The basic concept behind these utilities is to separate out the "process simulation" and "process analysis" parts of Suprem-III. Generally, one can use Suprem-III primarily to generate

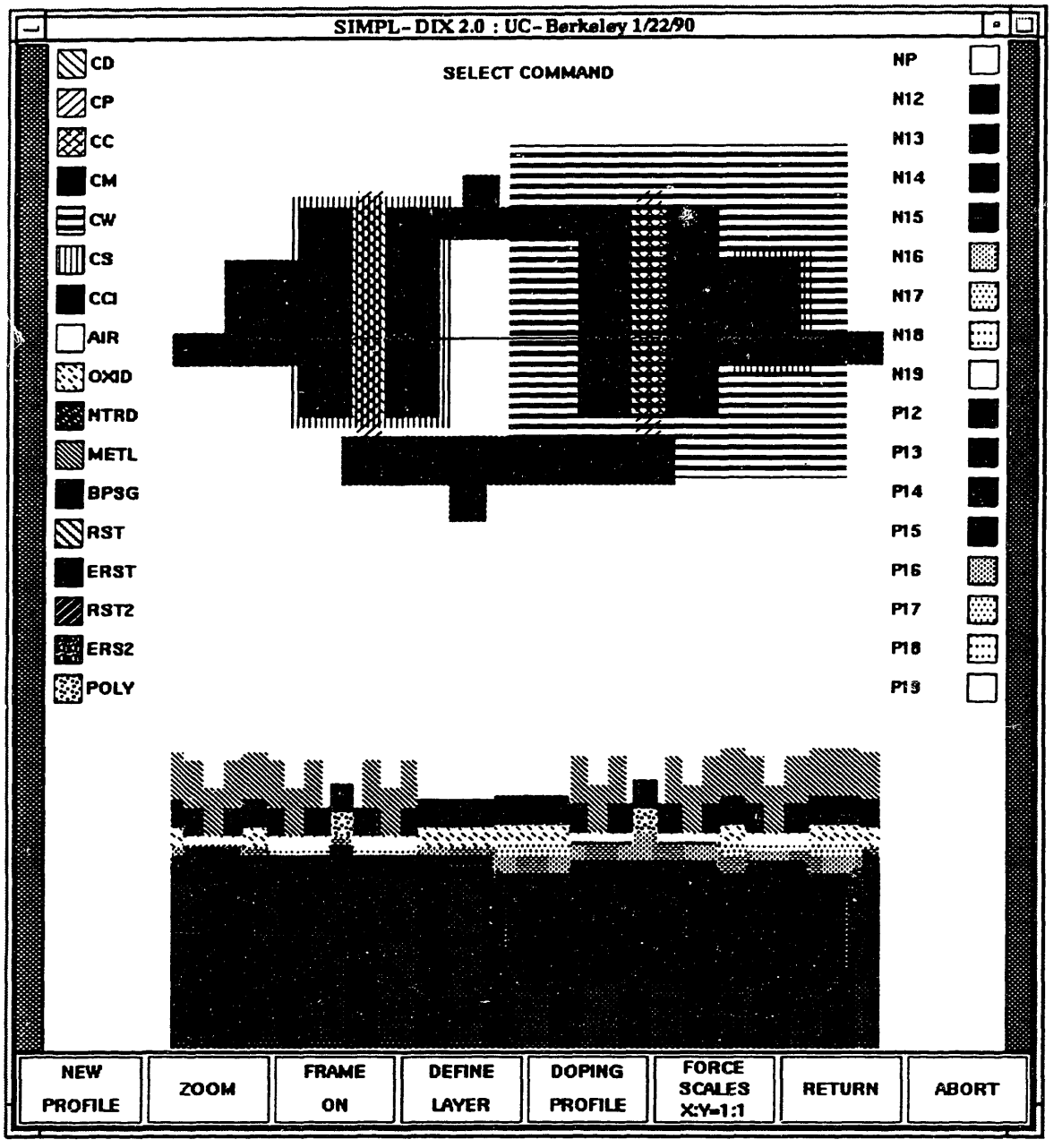


Figure 14.2: SIMPL-2 simulation of the CMOS baseline process.

the descriptions of the wafer structure as demanded by processing. That structure can be saved using the `Save Structure File=foo.sav` statement inside a Suprem-III input file, and then the commands below may access the save file to generate plots or to perform additional analyses.

The `layer` command takes as its argument the saved structure file (i.e. "nchan"), and uses Suprem-III to write layer thickness information to the standard output. The `sheetr` command takes as its argument the saved structure file, and uses Suprem-III to generate and write sheet resistance and layer conductivity and resistivity information to the standard output. The `vtn` and `vtp` commands take as it arguments the saved structure file, and uses Suprem-III to generate n-channel and p-channel threshold voltage information, respectively.

Plots of impurity concentrations may be generated for a number of output devices. The `supplot` command takes as a mandatory argument the name of the saved structure file, and generates an X-window system display. The `psplot` and `ps4bplot` commands generate hard copy output files suitable for printing on PostScript printers. The `screenplot` will generate output directly for "dumb" crt terminals. All of these plot commands use `postsup` to read the saved structure file, generate plotting files for `giraphe3`, invoke `giraphe3`, and then clean up the temporary plotting files. Additional arguments to these plotting commands are passed to `postsup`. For instance, `postsup nchan xmin=0` will plot beginning at the silicon surface. For more information on `postsup`, an on-line UNIX manual page has been written.

14.3.2 Baseline Process Analyses

For the analysis of the baseline process, two kinds of files are generated. First of these are `.char` (or "characterization") files that contain textual characterization information about each cross section, including layer thicknesses, sheet resistances, and threshold voltages (when appropriate). The second is a `.grp` (or "giraphe3") file that is used to generate a PostScript plot of the cross section. The script `characterize` in

Parameter	Cross Section	
	nchan	pchan
Poly thickness	0.45 μm	0.46 μm
Poly sheet R	19.8 $\Omega/\square$	42.0 $\Omega/\square$
Gate oxide thick	216 Å	216 Å
	ndrain	pdrain
Junction depth	0.37 μm	0.59 μm
Drain sheet R	38.5 $\Omega/\square$	67.0 $\Omega/\square$
	nfield	pfield
Field ox thick	0.45 μm	0.45 μm

Table 14.5: CMOS baseline process simulation results.

Figure 14.3 shows the commands that are used to perform these characterizations for three of the six cross sections of interest in the CMOS baseline process. The results of these analyses for the CMOS baseline simulations are summarized in Table 14.5.

14.4 Device Simulation

The two-dimensional device simulator Minimos is used to calculate current, voltage, and other characteristics of an MOS device [95]. In order to obtain accurate device characteristics, it is necessary to perform two-dimensional device simulations, which in turn requires a 2D description of the impurity profiles. The generation of these 2D structures is described below, followed by a summary of the device simulations themselves.

14.4.1 2D Structure

The `supmin` program is used to "rotate" and "merge" 1D profiles into a 2D structure.† The script `sav2min` first converts the Suprem-III `.sav` file to a `.min` file that `supmin` is

†The supmin program was developed and modified by various workers at Digital Equipment Corporation and MIT, including Marden Seavey and Jarvis Jacobs.

```
#!/bin/csh
#
# A script to generate the characterization and plot files
# for the cmos baseline process.

# nchan Characterization
layer nchan >! nchan.char
sheetr nchan >> nchan.char
vtn nchan >> nchan.char
postsup nchan xmin=0 xmax=1 cmin=1e12 cmax=1e18 \
  boron active and arsenic active \
  and phos active /nocolor
giraphe3 nchan -t ps -o

# ndrain Characterization
layer ndrain >! ndrain.char
sheetr ndrain >> ndrain.char
postsup ndrain xmin=0 xmax=1 cmin=1e14 cmax=1e21 \
  boron active and arsenic active \
  and phos active /nocolor
giraphe3 ndrain -t ps -o

...

# pfield Characterization
layer pfield >! pfield.char
postsup pfield xmin=0 xmax=1 cmin=1e12 \
  cmax=1e18 boron active and arsenic active \
  and phos active /nocolor
giraphe3 pfield -t ps -o
```

Figure 14.3: Script used to characterize the final cross sections produced by the CMOS baseline process.

```
#!/bin/csh
#
# Create the 2D doping file
postsup nchan.sav
postsup ndrain.sav
sav2min nchan.sav 4
sav2min ndrain.sav 4
supmin << DONE
ndrain.min
ndrain.min
nchan.min
1.0
Y
0.7
nmos.junk
DONE
mv nmos.DOP nmos.dop
minimos nmos nmos
# Compare it with the originals
giraphe3 nchan-compare
giraphe3 ndrain-compare
```

Figure 14.4: Script to create 2D device structure from 1D Suprem-III simulations.

able to read. The second argument to `sav2min` is the distance from the surface, in microns, in which the device is assumed to lie. In this case, 4 microns is used so as to avoid any confusion over the underlying wells and epitaxial layers. The script in Figure 14.4 generates the nmos device structure. One of the last commands in that script, "`minimos nmos nmos`" runs a simple Minimos simulation whose primary purpose is to generate a file `nmos.doping` that contains the 2D profile as interpolated from the `nmos.dop` file produced by `supmin`. Two plots are then generated that compare these profiles through channel (Figure 14.5) and drain (Figure 14.6) sections. Differences between these profiles can result in discrepancies between simulated and actual device characteristics, and should be minimized. Finally, the Minimos input file used to generate the `nmos.doping` file is shown in Figure 14.7 (the `PIF` output statement is a local enhancement to Minimos†).

†The MINIMOS program used in this work has been enhanced substantially by Jarvis Jacobs at MIT. The 'PIF' output statement generates giraphe3 readable impurity profile data, and does not

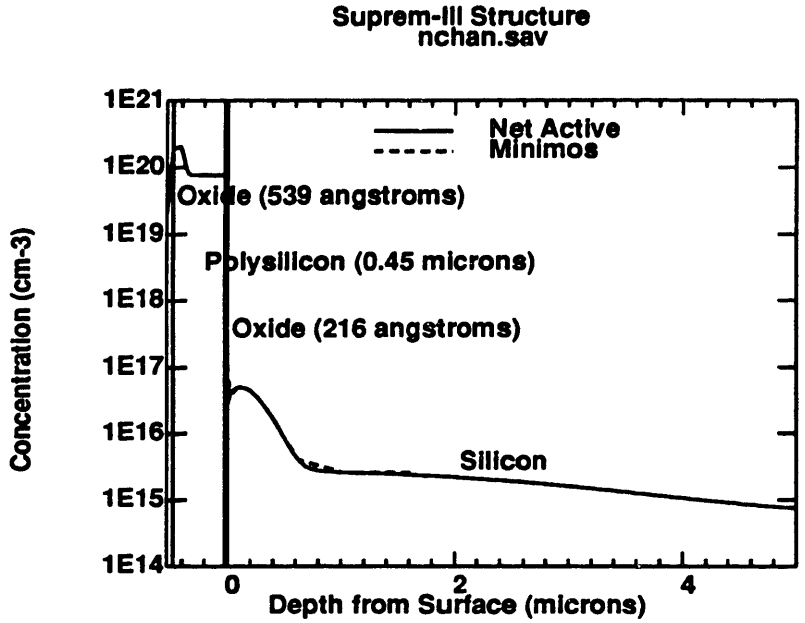


Figure 14.5: Comparison of nchan profile from Suprem-III and used by Minimos.

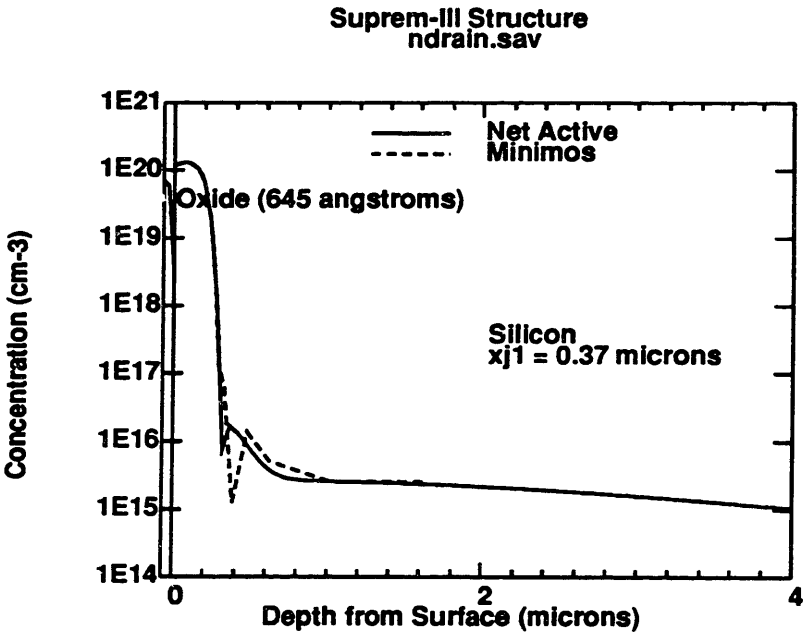


Figure 14.6: Comparison of ndrain profile from Suprem-III and used by Minimos.

```
TITLE
* TINS- TOX   DGAP - distance from gate to contact
* L- length
* W - width
DEVICE CHANNEL=N GATE=N TINS=216E-8 L=2.0E-4
+ W=20E-4 INS=OXIDE
+ DGAP=0.5E-4 SGAP=0.5E-4
*  US-source  UB- bulk  UD -drain  UG-gate
BIAS US=0 UB=0 UD=0.05 UG=1.5
PROFILE FILE=1 ASYM=Y LFIT=0.7
OPTION MODEL=2 PHYSCK=N
+TEMP=300 GRIDFREEZE=N
PIF DC=Y
OUTPUT ALL=Y
END ERROR=1E-3 BIN=N TERR=1E-3
```

Figure 14.7: The Minimos input file used to read `nmos.dop` and generate `nmos.doping` files.

14.4.2 Minimos Simulations

Three different kinds of simulations are run: a threshold voltage calculation, a subthreshold characterization, and a family of drain-current simulations. The script that performs these simulations is shown in Figure 14.8. The various Minimos input files used for that calculation are shown in Figures 14.9, 14.10, and 14.11. The two dimensional doping profile for the NMOS device is shown in Figure 14.12. The threshold voltage calculations for the NMOS device are shown in Figure 14.13, the subthreshold characteristics in Figure 14.14, and the current output characteristics in Figure 14.15. Similar plots for the PMOS device are shown in Figures 14.16, 14.17, 14.18, and 14.19.

14.5 Summary of Case Study

The methods used to analyze the CMOS baseline process have been described. The process has been written using the MIT Process Flow Representation, including de-

---

actually output Profile Interchange Format compatible data (despite the statement name).

```
#!/bin/csh
#

# Vt's
minimos nmos-vt nmos
echo ".col source gate drain 1d-current 2d-current" \
  >! nmos-vt.data
grep DATA nmos-vt.out | \
  sed -e "s/DATA//" >> nmos-vt.data
giraphe3 nmos-vt-plot -t ps -o

# Subthreshold Vt
minimos nmos-subvt nmos
echo ".col source gate drain 1d-current 2d-current" \
  >! nmos-subvt.data
grep DATA nmos-subvt.out | \
  sed -e "s/DATA//" >> nmos-subvt.data
giraphe3 nmos-subvt-plot -t ps -o

# Id's
minimos nmos-id nmos
echo ".col source gate drain 1d-current 2d-current" \
  >! nmos-id.data
grep DATA nmos-id.out | \
  sed -e "s/DATA//" >> nmos-id.data
giraphe3 nmos-id-plot -t ps -o
```

Figure 14.8: Script used to perform Minimos simulations.

```
TITLE
* TINS- TOX   DGAP - distance from gate to contact
* L- length
* W - width
DEVICE CHANNEL=N GATE=N TINS=216E-8 L=2.0E-4
+ W=20E-4 INS=OXIDE
+ DGAP=0.5E-4 SGAP=0.5E-4
*  US-source  UB- bulk  UD -drain  UG-gate
BIAS US=0 UB=0 UD=0.05 UG=0.5
STEP  NG=15  DG=0.1  DD=0.0 ND=0
PROFILE FILE=1 ASYM=Y LFIT=0.7
* RECOM AN=0 AP=0 CN=0 CP=0
OPTION MODEL=2 PHYSCK=N
+TEMP=300 GRIDFREEZE=N
* PIF DC=Y
OUTPUT ALL=N NONE=Y
END ERROR=1E-3 BIN=N TERR=1E-3
```

Figure 14.9: Minimos input file `nmos-vt.inp` for threshold calculation.

```
TITLE
* TINS- TOX   DGAP - distance from gate to contact
* L- length
* W - width
DEVICE CHANNEL=N GATE=N TINS=216E-8 L=2.0E-4
+ W=20E-4 INS=OXIDE
+ DGAP=0.5E-4 SGAP=0.5E-4
*  US-source  UB- bulk  UD -drain  UG-gate
BIAS US=0 UB=0 UD=0.05 UG=0.1
STEP  NG=19  DG=0.1  DD=0.0 ND=0
PROFILE FILE=1 ASYM=Y LFIT=0.7
OPTION MODEL=2 PHYSCK=N
+TEMP=300 GRIDFREEZE=N
OUTPUT ALL=N NONE=Y
END ERROR=1E-3 BIN=N TERR=1E-3
```

Figure 14.10: Minimos input file `nmos-subvt.inp` for subthreshold characterization.

```
TITLE
* TINS- TOX   DGAP - distance from gate to contact
* L- length
* W - width
DEVICE CHANNEL=N GATE=N TINS=216E-8 L=2.0E-4
+ W=20E-4 INS=OXIDE
+ DGAP=0.5E-4 SGAP=0.5E-4
*  US-source  UB- bulk  UD -drain  UG-gate
BIAS US=0 UB=0 UD=0.5 UG=1.0
STEP  NG=4  DG=1.0  DD=0.5 ND=9
PROFILE FILE=1 ASYM=Y LFIT=0.7
OPTION MODEL=2 PHYSCK=N
+TEMP=300 GRIDFREEZE=N
OUTPUT ALL=N NONE=Y
END ERROR=1E-3 BIN=N TERR=1E-3
```

Figure 14.11: Minimos input file `nmos-id.inp` for output characteristics.

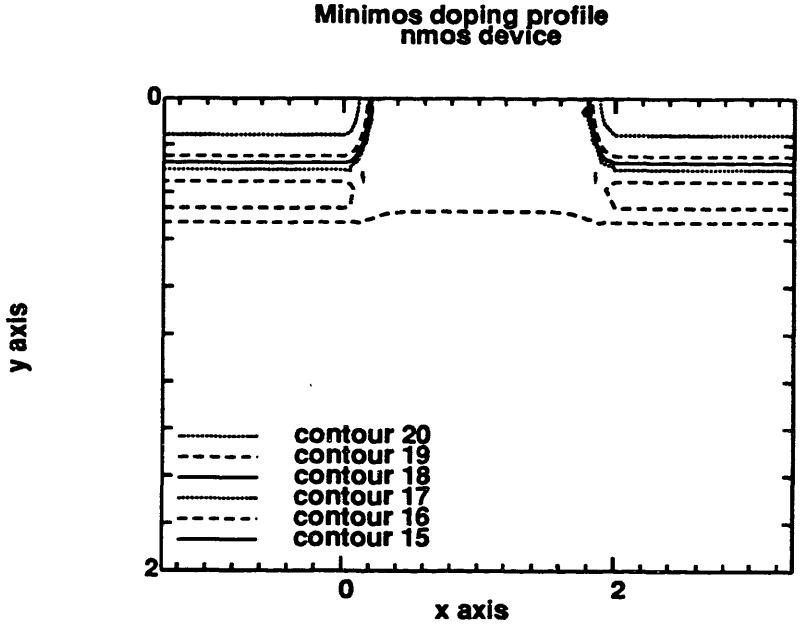


Figure 14.12: NMOS Two Dimensional Doping Profile

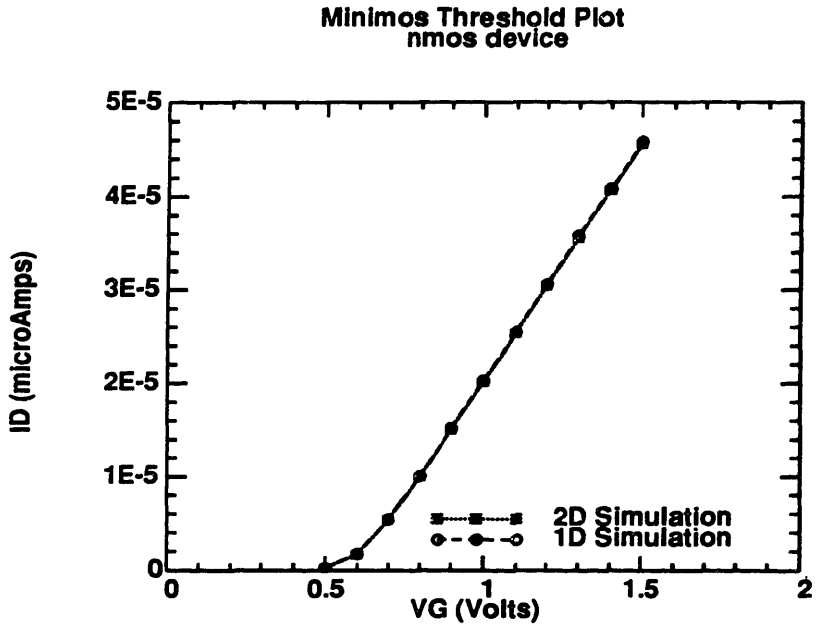


Figure 14.13: NMOS Threshold voltage

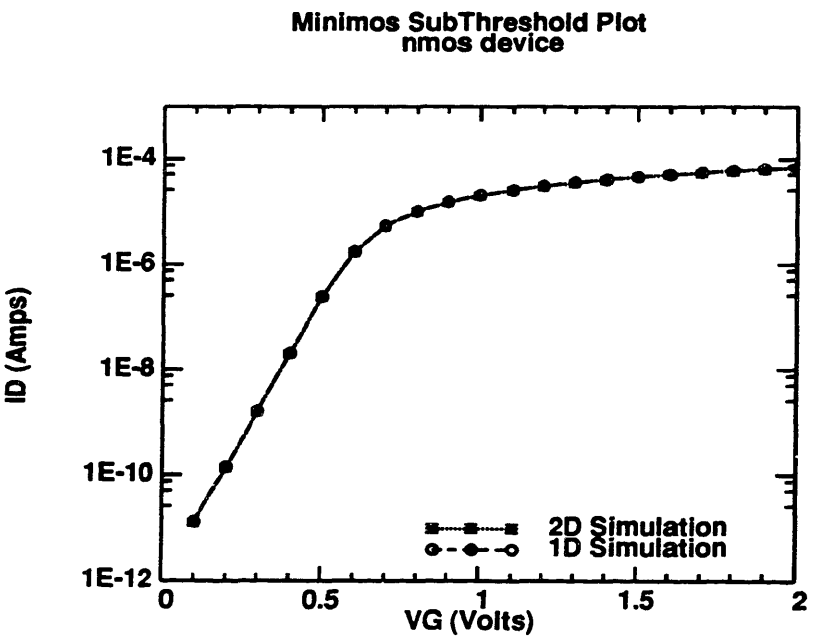


Figure 14.14: NMOS Subthreshold characteristics

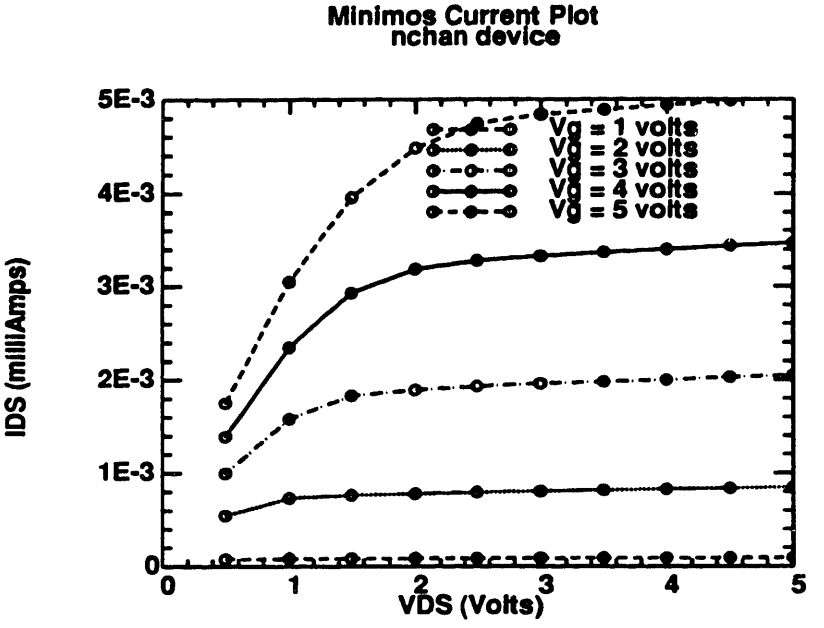


Figure 14.15: NMOS Current-Voltage characteristics

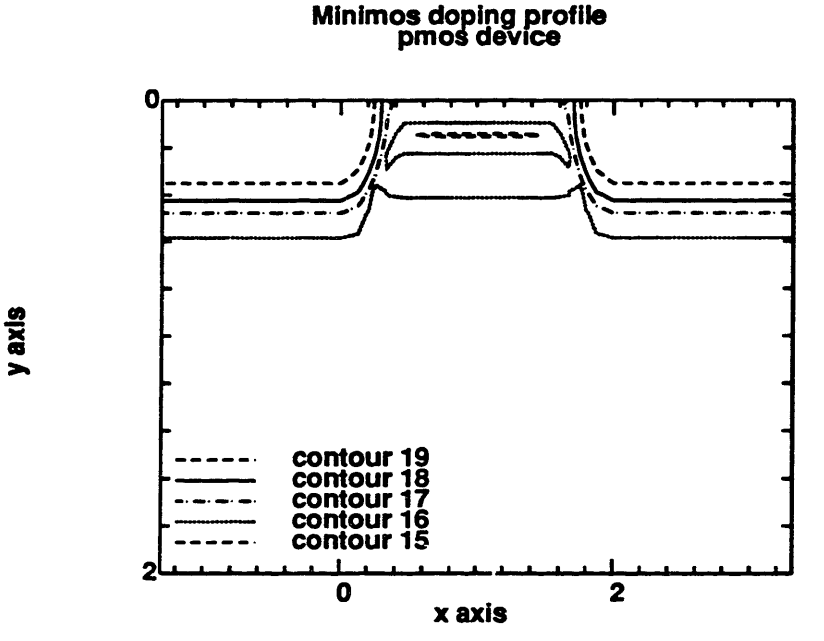


Figure 14.16: PMOS Two Dimensional Doping Profile

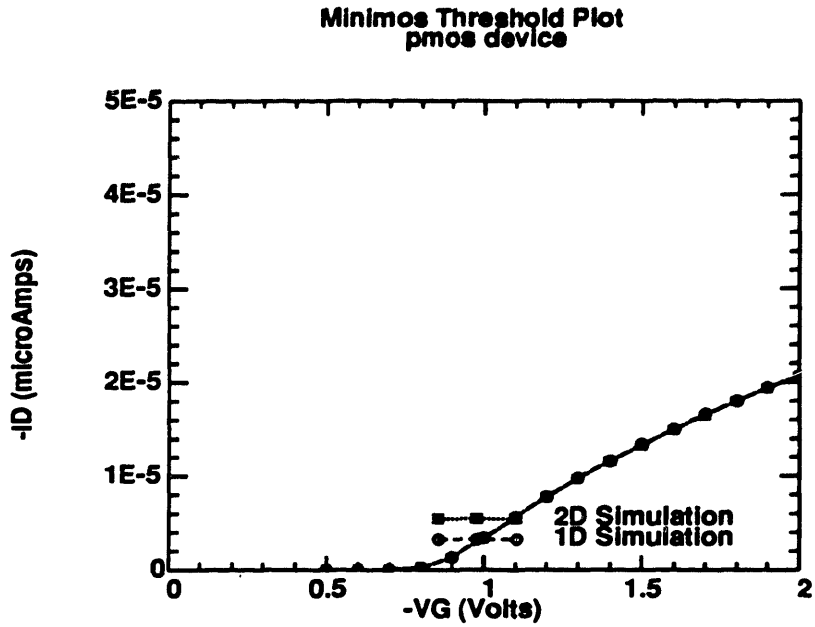


Figure 14.17: PMOS Threshold voltage

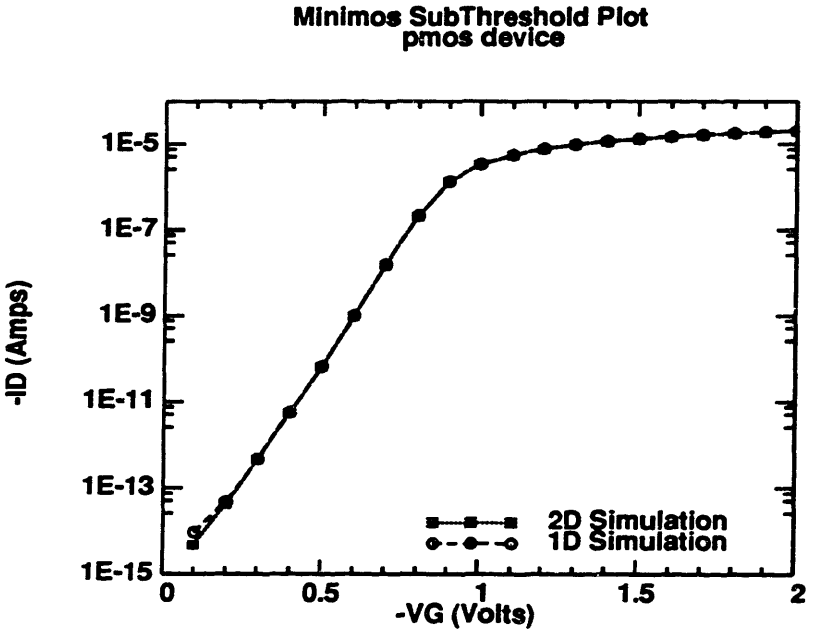


Figure 14.18: PMOS Subthreshold characteristics

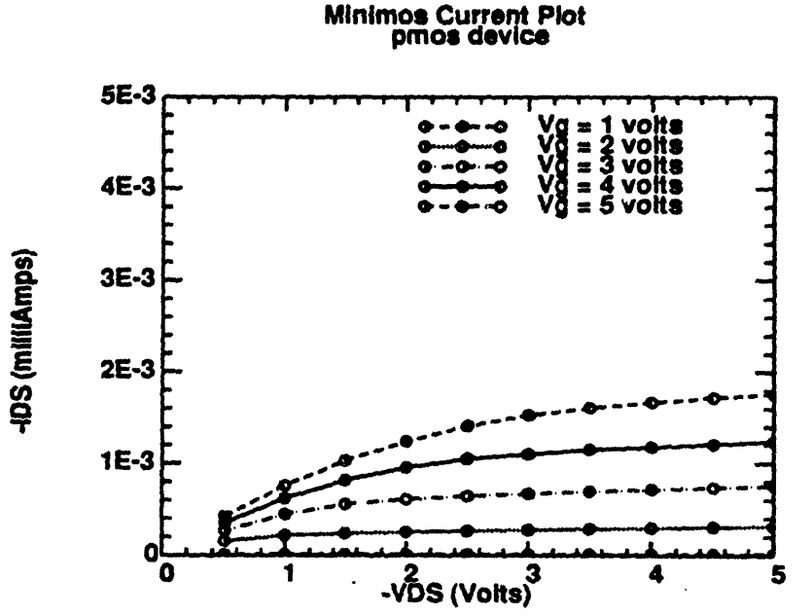


Figure 14.19: PMOS Current-Voltage characteristics

scriptions of the masks and cross sections of interest. The process treatment was simulated in one dimension using Suprem-III via the Simulation Manager, and the change in wafer state was simulated in two-dimensions using the SIMPL-2 translator and SIMPL-2. The commands and scripts used to characterize the resulting wafer structures have been presented. The programs used to generate and check a two-dimensional description of the device has been shown, and the input files and commands necessary to perform and plot device simulations shown as well.

This detailed summary of the methods used to perform such process and device simulations should be of help to anyone who is attempting to simulate, analyze, or develop semiconductor processes using the tools developed as part of this thesis. While some steps in this procedure are relatively automated (such as those involved in direct process simulation), many others in the analysis of wafer and device structures are not yet automated. A language (or script methodology) for describing the invocation of TCAD tools and the manipulations of data in an integrated TCAD Framework would greatly ease this and other problems in technology development and analysis [10].

Chapter 15

Case Study: Baseline Process Enhancement

An enhancement to the MIT CMOS baseline process has been performed in order to demonstrate the design, transfer, and execution of an application specific fabrication process. The baseline process has been augmented to include a poly-to-silicon capacitor, and illustrates in a simple test case (1) the design representations, tools, and methodologies discussed in this thesis, (2) the transfer of information to the fabrication facility sufficient to perform the enhanced process, and (3) the execution of that process.

15.1 Experiment Goals

The device goal is to enhance the existing CMOS baseline process to make available a floating capacitor with low voltage coefficient and large capacitance per area. The enhancement should have minimal impact on other structures produced by the pre-existing baseline process. Such a capacitor might be suitable in analog circuit applications. The research goal is to illustrate the following:

1. Demonstrate design capability by

(a) using the process representation to describe the process

(b) using tools, including the simulation manager and process advisors, to both design the basic device and to examine the impact on the baseline process

(c) using the methodologies for single device design (including process rough-in, parameter estimation, and parameter optimization) and process integration.

2. Demonstrate the transmission of the process to manufacturing by

   (a) producing manufacturing-compatible descriptions of the process (traveler and runsheet)

   (b) producing mask information

   (c) submitting the process to a fabrication facility

3. Demonstrate application specific fabrication by

   (a) executing the process in the facility as specified

   (b) showing that the modified process produces working devices on "first silicon".

There are limits to the capacitor enhancement and the conclusions that can be drawn from this test case. First, this is an example of an enhancement to an existing process (and one in which enhancements such as this were intended at initial conception), and is not an radically new process. Second, the modification adds a relatively simple device to the process, and results in comparatively minor modifications to the process. Finally, the PFR is used only in design of the process and not during actual fabrication, so that additional conversion of process information is necessary. Nevertheless, this experiment remains a non-trivial demonstration of the concept of application specific processing, and more specifically, the use of representations, tools, and methodologies in the design of such processes.

15.2 Process Design

15.2.1 Rough-In

The abstract structure of the desired capacitor is shown in Figure 15.1. Silicon will

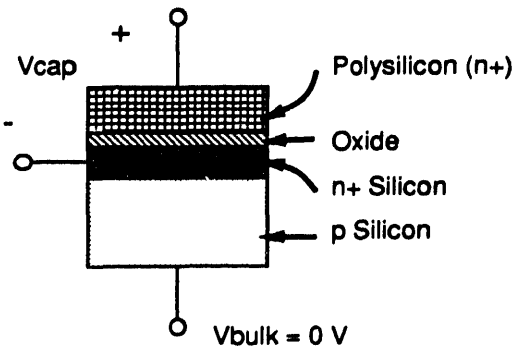


Figure 15.1: Abstract poly-to-silicon capacitor structure.

be doped heavily to form the bottom plate of the capacitor, a thin oxide will serve as the dielectric, and a heavily doped polysilicon layer will form the top plate. The primary goal of the structure is to achieve a low capacitor voltage coefficient (V_{cc}), defined as

$$V_{cc} = \frac{1}{C}\frac{dC}{dV} \tag{15.1}$$

where C is the total device capacitance and V is the voltage applied across the device [96, 97]. This MOS structure consists of an oxide capacitor in series with a depletion capacitor. The similarity to the gate region of an MOS transistor points out the essential design problem in using the structure as a capacitor. In the MOS transistor, application of a voltage across the structure is *intended* to deplete and invert the channel region, so that there is a large dependency on the depletion layer capacitor and thus on the voltage across the device. A high surface doping concentration in the silicon will minimize the depletion layer width dependence on voltage, and thus result in a nearly constant capacitance with respect to voltage.

Other design choices must be made. An $n+$ bottom plate in the p-well is chosen so that the bottom plate will be doped similarly to the top plate, and depletion capacitances in both layers will help to cancel each other out [97]. The use of arsenic or phosphorus remains a design parameter.

Working from the substrate upward, the capacitor structure directly suggests an

essential mutator sequence: dope silicon, grow oxide, and deposit polysilicon.

15.2.2 Process Integration

The next step is to search for ways to integrate or merge the basic capacitor mutator sequence with the existing baseline process. Two kinds of merges are possible: *sharing* with existing process steps, and *insertion* into the process sequence. The baseline process produces two oxide layers – the field and channel regions, and one polysilicon layer. The capacitor can share the polysilicon and gate oxide layers (the thick field oxide would result in a capacitance that is much too small). Thus, a merge of the baseline and capacitor mutator sequences suggests that the ion implant should take place before the growth of the gate oxide in order to share the gate and polysilicon steps.

The *insertion* of the implant step is slightly more complicated. The silicon in the capacitor should be heavily doped, but the other structures on the wafer must not be affected. That is to say, a *selective* implantation is necessary, so that the implant will be encapsulated with a selection mutator. The canonical implementation of the selection mutator – via a photolithography, the implant, and a resist removal – is sufficient to decouple this implant from the other structures on the wafer. The selected implant (including the photolithography) can now be inserted at any point in the process where silicon (or thin oxides) are present. It is preferable to implant through a thin oxide in order to scatter the implanted ions, suggesting possible positions for the implant in the sequence: (1) during the p-well formation, just after or before the p-well ion implant, where the original stress relief oxide has been exposed; (2) between the active area and field formation operations, where implantation through both a stress relief oxide and a nitride layer would be necessary, and (3) between the dummy gate oxide growth and its etch during the channel formation. The close similarity between the MOS transistor and the proposed MOS capacitor suggests the last choice as a good position for the capacitor implant in the overall process, though other positions

Surface Concentration (cm^{-2})	oxide thickness (Å)
1×10^{18}	104
1×10^{19}	111
2×10^{19}	122
4×10^{19}	151
8×10^{19}	224
1×10^{20}	264
2×10^{20}	434
4×10^{20}	588
8×10^{20}	670
1×10^{21}	686

Table 15.1: Relationship between surface concentration and resulting capacitor oxide thickness as predicted by the oxidation Advisor.

could also be explored (the primary difference being the amount of diffusion and oxidation the capacitor sees before the growth of the gate/capacitor oxide layer). The final choice is to perform the selected implant following the n and p device channel implants, and before the etch of the dummy gate oxide.

15.2.3 Parameter Estimation

The next step is the estimation of the important parameters for the capacitor process steps. Because the merged process uses the poly and oxide growths from the existing process, no additional modification of these parameters is possible. However, intuitive understanding of the effect of the process on the capacitor structure is important. First, the geometry of the resulting structure must be examined. For a one-dimensional view, the Process Advisor can be used to explore the oxide thicknesses that will result from different concentrations of implanted impurities in the silicon. For a 950C, 30 minute dry oxidation, the oxide thicknesses predicted by the oxidation Advisor are shown in Table 15.1, as produced using the command line interface and the script of Figure 15.2. The table illustrates the essential tradeoff in the

```
foreach conc ( 1e18 1e19 2e19 4e19 8e19 1e20 2e20 4e20 8e20 1e21 )
    set tox = 'oxidation bg_conc=1e15 sf_impurity=phos sf_conc=conc \
        dry temp=950 time=30 action=analytic'
    echo conc tox
end
```

Figure 15.2: Script used to produce results shown in Table 15.1.

capacitor process: the higher the implant dose and surface concentration, the lower will be V_{cc}, but also the thicker will be the oxide (and thus the lower the C_{ox}, where $C_{ox} = \frac{\epsilon_{ox}}{t_{ox}}$ is the oxide capacitance per unit area, ϵ_{ox} the permittivity of the oxide, and t_{ox} the thickness of the oxide). This tradeoff must be explored further during parameter optimization. For now, it suffices to estimate ranges for the remaining process parameters. The implant impurity is limited to arsenic or phosphorus, and V_{cc} will be minimized by variation of the implant dose and energy while maintaining an acceptable oxide capacitance.

15.2.4 Process Optimization

The optimization loop for determination of impurity type, energy, and dose requires process simulation, device simulation, and parameter extraction as summarized in Figure 15.3. One dimensional process simulation is performed using Suprem-III [24]. The input file is generated from a complete description of the modified baseline process written in the PFR. This input file is then modified iteratively by the design space exploration script, which encapsulates the tool invocations, conversions, and extractions. One dimensional device simulation is performed using Sedan-III [98], which takes as input the doping profile in the silicon and the oxide thickness, and produces oxide and total MOS capacitances as a function of bias voltages. V_{cc} is finally calculated from the Sedan-III output files, and the results are plotted using Giraphe3 [72].

Because the simulations are relatively inexpensive computationally, a simple ex-

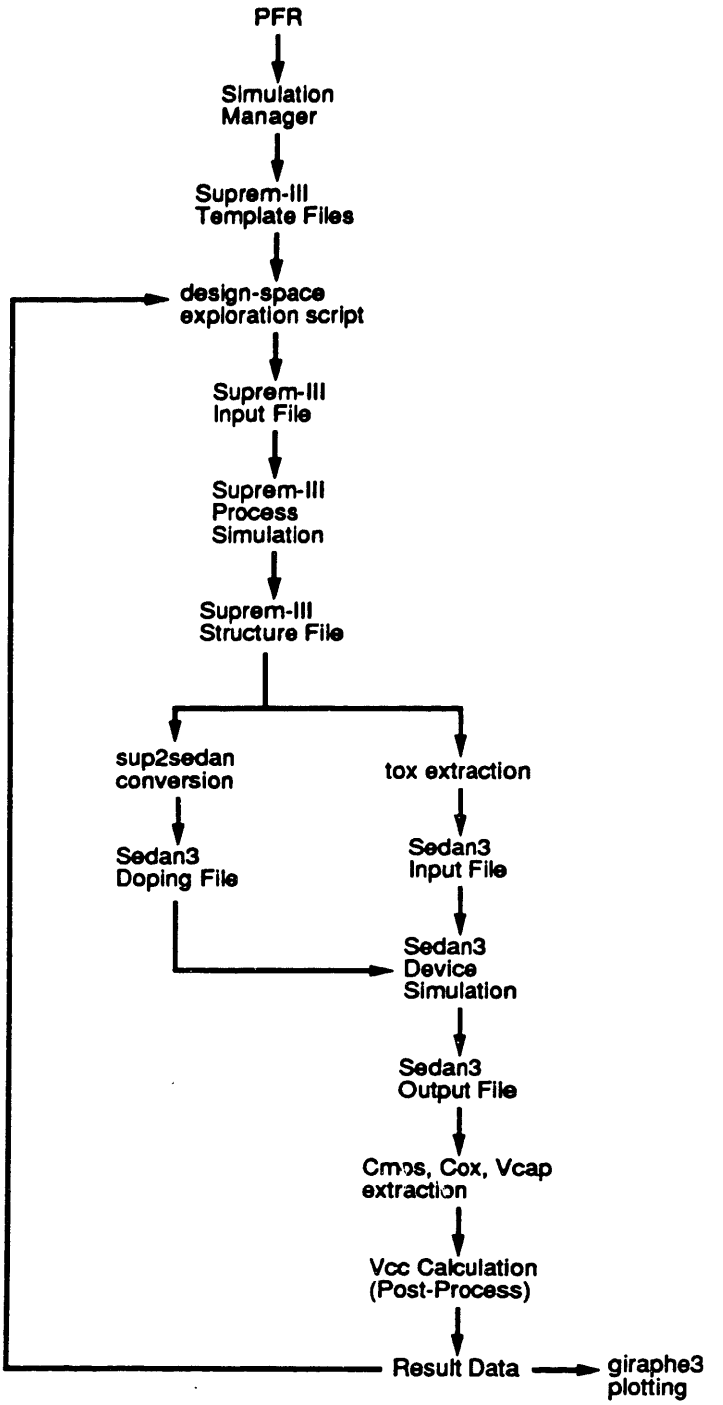


Figure 15.3: Optimization flow in determination of optimal capacitor design parameters.

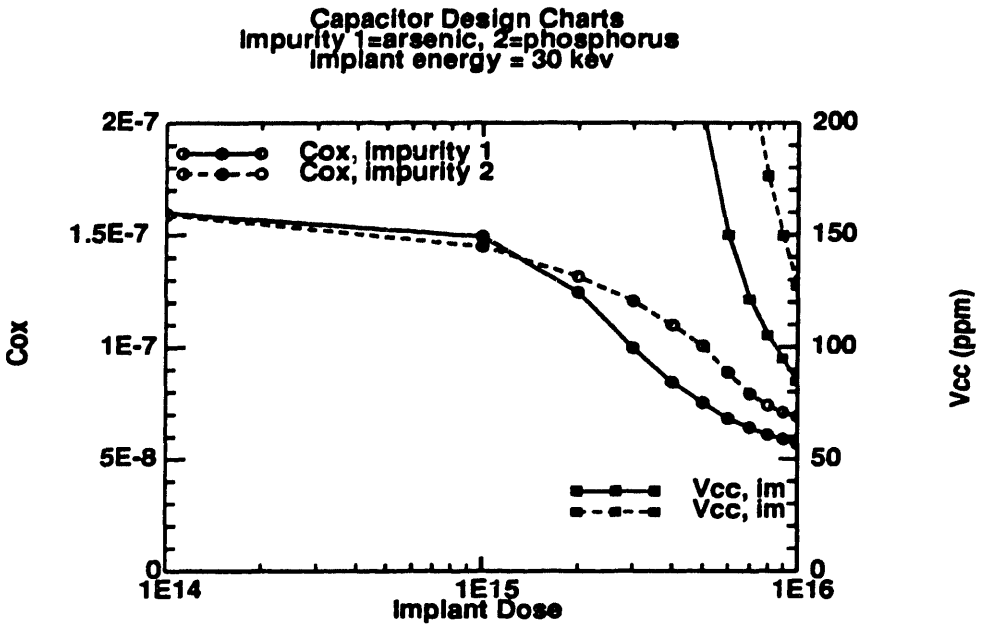


Figure 15.4: C_{ox} and V_{cc} as a function of implant dose for arsenic and phosphorus implants at 30 kev.

perimental design varying arsenic, phosphorus, implant energy, and implant dose independently can be performed. For more costly simulations, designed experiments to reduced the number of simulations required would be appropriate [99].

Based on the simulated results, the implanted dose should be chosen as high as possible. As shown in Figure 15.4, the voltage coefficient monotonically decreases with implant dose. The dose is limited to 1×10^{16} cm^{-2} in order to satisfy the guidelines of the fabrication facility. The dependence of V_{cc} and C_{ox} on implant energy and dopant for this dose is summarized in Figure 15.5. First, the voltage dependence for arsenic implants is nearly 50% less than for phosphorus and the oxide capacitance is 35–40% less, so that doping with arsenic minimizes V_{cc} while maintaining an acceptable C_{ox}. In this case, the minimal V_{cc} is achieved for an implant energy of between 60 and 70 kev, and 65 kev is chosen to minimize the sensitivity of V_{cc} with respect to implant energy. The resulting doping profile for the capacitor as simulated by Suprem-III is shown in Figure 15.6. Simulations using SIMPL-2 (with input generated from the

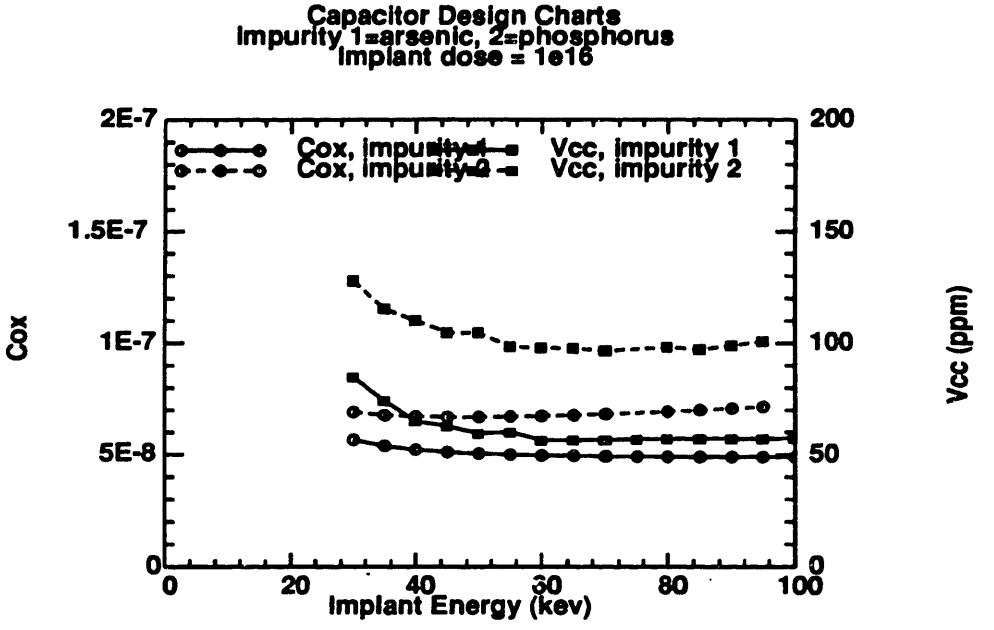


Figure 15.5: C_{ox} and V_{cc} as a function of implant energy for arsenic and phosphorus implants at dose 1×10^{16} cm^{-2}.

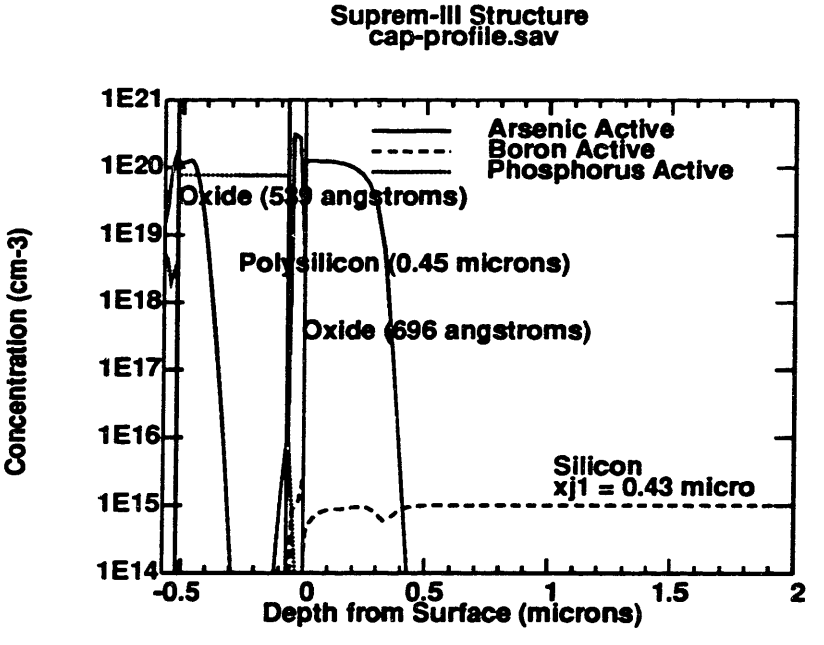


Figure 15.6: Doping profile for baseline poly-to-silicon capacitor.

PFR by the SIMPL-2 translator) are shown in Figures 15.7 and 15.8. Figure 15.7 shows the overall layout of the capacitor test structure†, and a cross section with the two contacts to the top capacitor plate on the right, and contact to the bottom capacitor plate (the thin, highly doped diffused region) on the left. Figure 15.8 shows a cross section with two contacts to the top capacitor plate on the right, and contact to the well on the left (where the well contact is isolated from the capacitor bottom plate by a field oxide).

15.3 Process Transmission

The transmission of the process enhancements to the fabrication facility are in two parts. First, the process information determined from the design of the process must be put in the form understood by the manufacturing system itself. Secondly, information sufficient to produce the masks for the baseline capacitor lithography step must be produced and transmitted.

15.3.1 Process Description

The PFR is used for describing the process during the design of the baseline capacitor enhancement. CAFE is intended to support fabrication directly based on the PFR [15] and a first attempt at experimental use of the PFR in this capacity is underway. For the purposes of the baseline enhancement, however, the PFR-based parts of CAFE are not used, both to decouple the PFR experiment from the enhancement demonstration, and to demonstrate further the power of a single process description in coupling design to *existing* manufacturing systems.

The opset and traveler for the baseline process enhancements are generated automatically using the programs described in Chapter 11. These descriptions of the process were approved by a standing process technology policy committee, and then

†The layouts were done by Prabha Tedrow.

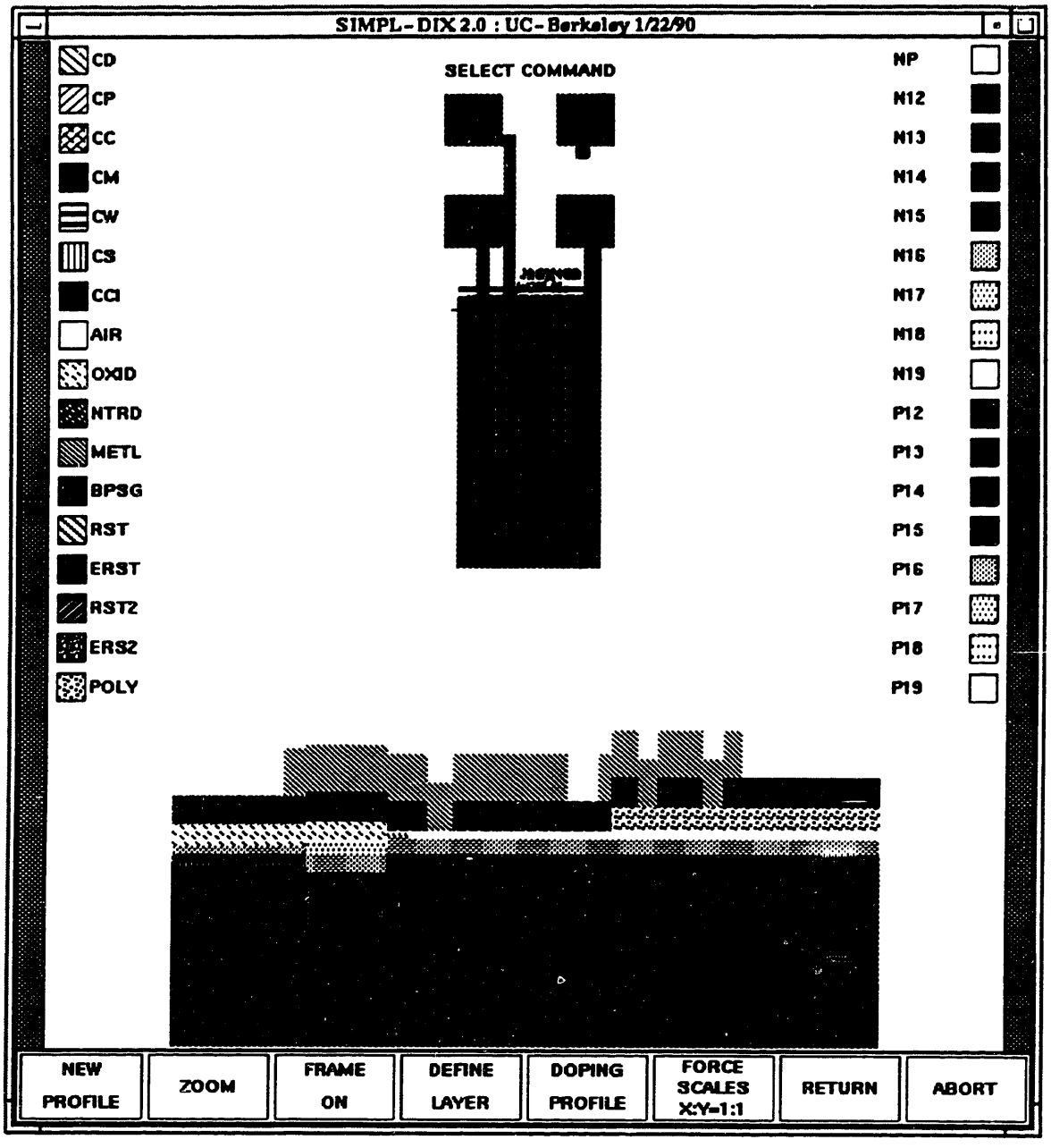


Figure 15.7: SIMPL-2 simulation of the capacitor enhancement to the CMOS baseline process.

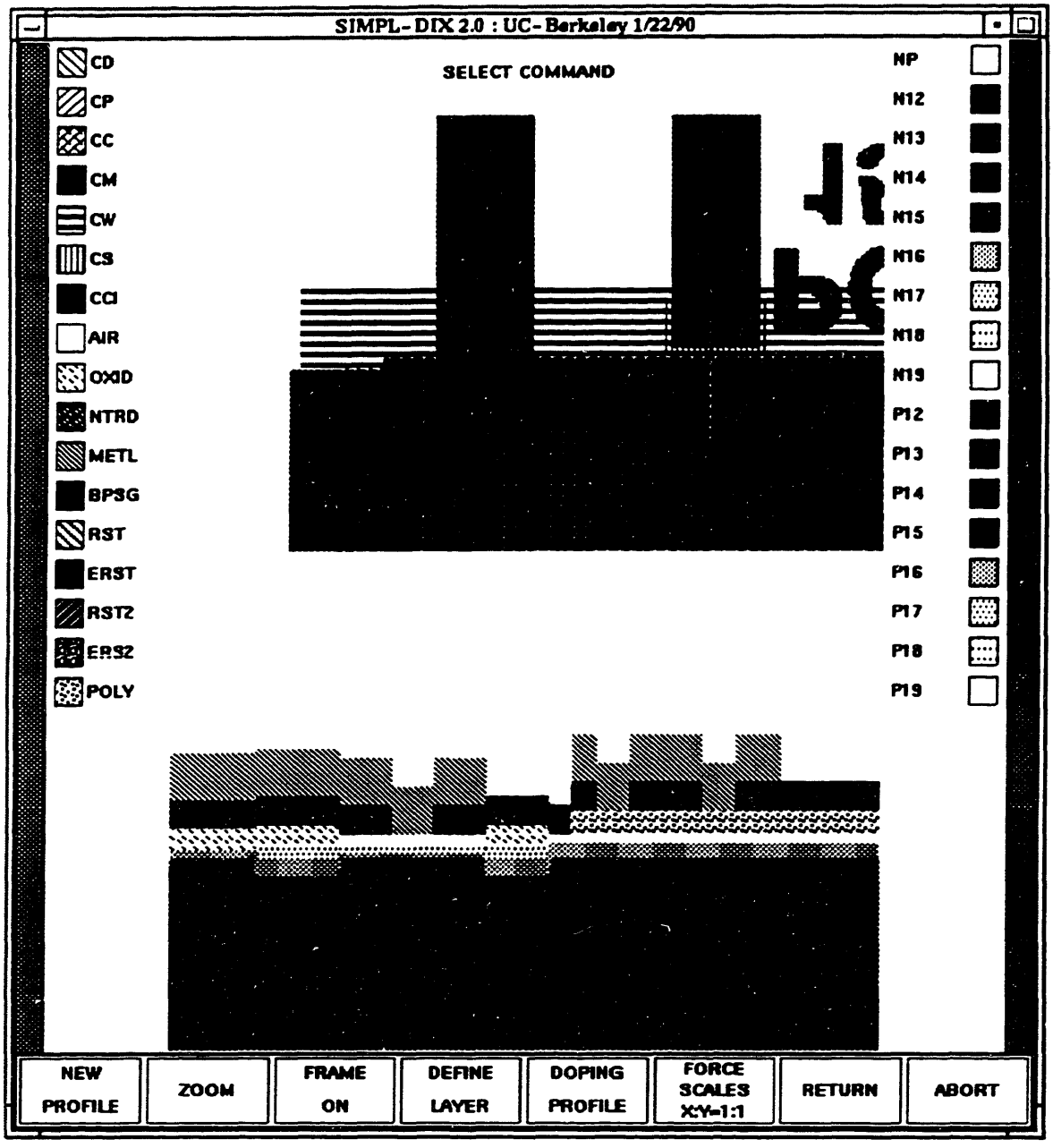


Figure 15.8: SIMPL-2 simulation of the capacitor enhancement to the CMOS baseline process.

electronically mailed to the engineer in charge of the execution of the defect array lot (one of which is started on a recurring schedule to monitor the execution of the baseline process in the MIT Integrated Circuits Laboratory), who then inserted the additional traveler lines into the existing traveler for lot DA30.

15.3.2 Mask Description

The test capacitor resides in the baseline drop-in test pattern used in all ICL baseline runs, including defect array lots. The original drop-in incorporated layout layers for a single capacitor structure, and for contact to the capacitor [94]. This greatly eased the burden of mask-making; the active-area, poly, and metal masks did not have to be modified or regenerated, as they already contained the necessary structures. The mask for the capacitor implant, however, had to be made. The basic layout layer for the capacitor implant consists of a single box lying within the capacitor active area. To this are added alignment marks and blocking edges, optical patterns for evaluation of the lithography during processing, and verniers for aiding and measuring alignment. This combined layout was then converted to the appropriate representation for the generation of optical masks, and submitted to the facility for mask generation. An interesting observation arising from this experience is that the generation of physical masks for use in processing can itself be a complicated manufacturing exercise, and the adaptation of a process flow representation to aid in the generation of mask product rather than wafer product would be straight-forward and highly useful.

15.4 Fabrication

The enhancements to the baseline process were executed by the staff of the MIT Integrated Circuits Laboratory. Prolonged downtime of the CVD metal system, however, prevented completion of the DA30 defect array lot at this writing.

15.5 Conclusions

The enhancement of the MIT CMOS baseline process to incorporate a polysilicon-to-silicon capacitor demonstrates in a limited fashion the viability of the application specific processing concept. The availability of a single uniform representation of the process, tools for gaining intuitive, qualitative, and quantitative understanding of the process, and methodologies for the generation and integration of processes all ease the difficulty of designing and modifying a fabrication process. Tools to aid in the transmission and transfer of the resulting process and mask information are also important, as are information systems to manage the execution of the processes within a facility.

The baseline capacitor is a useful enhancement to a process. Many of the design decisions made in this experiment, particularly in the tradeoff between voltage coefficient and capacitance, would be different depending on the intended use of the capacitor within an analog circuit. That is, the choice of capacitor design parameters, and indeed of the capacitor structure itself, *depend on the intended circuit application*, and the resulting process is a true example of an application specific process.

Chapter 16

Conclusions

The essential contribution of this thesis is to identify, investigate, and prototype key software elements that will be necessary to support the design of application specific processes (ASPs). The integration of process design and manufacturing to a much greater extent than now occurs is recognized to be necessary. New methodologies, tools, and representations to support the automation of process design are proposed.

A methodology whereby descriptions of design data, including device characteristics, wafer structure, mutator sequences, treatment descriptions, and machine settings are generated and manipulated has been introduced. This thesis proposes *mutators* and mutator manipulation procedures as a methodology for the support of structure implementation and process integration.

This thesis offers an example of a tool for process design that is fundamentally different in nature than traditional simulation tools. These *Process Advisors* provide simple yet direct and useful support in the synthesis of process treatment information. This thesis also contributes a *Simulation Manager* to improve the interaction and effectiveness of process simulation.

This thesis has contributed to the development and understanding of both wafer and process description. First, an object-oriented approach to the conceptual modeling of wafer structures is proposed, the generation of a PIF toolkit and program

interface for use by application tools has been demonstrated, and small applications using the toolkit have been prototyped. Second, this thesis has contributed to the evolution of understanding and modeling of semiconductor processes via a generic process model, and to the evolution of the Process Flow Representation (PFR) to support the representation of the data used and manipulated during fabrication process design. Several kinds of tools based on the PFR have been prototyped, including design rule checkers, manufacturing specification generators, and process simulation interfaces.

These representations, tools, and methodologies for semiconductor process design do not by themselves enable the design of application specific processes. They do, however, lay the foundation and begin the long job of construction that lies ahead. We can evolve and construct the CAD and CIM software systems necessary to support the design and execution of application specific processes. Once we do so, the innovation in electronic and mechanical structures may well be boundless.

Bibliography

[1] D. S. Boning, "MASTIF – A workstation approach to integrated circuit process and device design," Master's Thesis, Massachusetts Institute of Technology, May 1986.

[2] D. S. Boning and D. A. Antoniadis, "MASTIF — A workstation approach to fabrication process design," *IEEE International Conf. on CAD, ICCAD-85*, pp. 280–282, Nov. 1985.

[3] D. S. Boning and D. A. Antoniadis, "A workstation approach to ic process and device design," *IEEE Design And Test of Computers*, pp. 36–47, Apr. 1988.

[4] R. T. Howe, R. S. Muller, K. J. Gabriel, and W. S. N. Trimmer, "Silicon micromechanics: sensors and actuators on a chip," *IEEE Spectrum*, pp. 29–35, July 1990.

[5] "Research in Microsystems Technology," Annual Report, MIT Microsystems Research Laboratories, 1990.

[6] M. B. McIlrath, D. E. Troxel, D. S. Boning, M. L. Heytens, P. Penfield Jr., and R. Jayavant, "CAFE: The MIT Computer-Aided Fabrication Environment," in *Proceedings of the International Electronics Manufacturing Technology Symposium*, (Washington, D.C.), Oct. 1990.

[7] J. Y. Pan, J. M. Tenenbaum, and J. Glicksman, "A framework for knowledge-based computer-integrated manufacturing," *IEEE Trans. Semiconductor Manufacturing*, vol. 2, no. 2, pp. 33–46, May 1989.

[8] M. Schroth private communication, July 1990.

[9] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, Massachusetts: Addison-Wesley, 1980.

[10] D. S. Boning, Editor, "TCAD Framework Architecture," SRC Publication P90013, Semiconductor Research Corporation, May 1990. (Also available as a CFI TCAD Framework Group document).

[11] M. B. McIlrath and D. S. Boning, "Integrating process design and manufacture using a unified process flow representation," in *Proc. Second Intl. Conf. on Computer Integrated Manufacturing*, (Troy, NY), pp. 224–230, IEEE Computer Society Press, Los Alamitos, CA, May 1990.

[12] R. Jayavant, "Programmer's Guide to Fabform," VLSI Memo No. 88–487, MIT, Sept. 1988.

[13] M. L. Heytens and R. S. Nikhil, "GESTALT: An Expressive Database Programming System," VLSI Memo No. 88–484, MIT, Nov. 1988.

[14] R. Jayavant, "An Intelligent Process Flow Language Editor," VLSI Memo No. 88-475, MIT, Sept. 1988.

[15] D. Troxel and M. B. McIlrath, "The MIT PFR – Application to fabrication," in *1990 DARPA-SRC Workshop on CIM for Integrated Circuits*, (U.C. Berkeley, Berkeley, CA), Aug. 1990.

[16] A. Kashani, "A reservations-based scheduler," in *Proceedings of the SRC Workshop on Integrated Factory Management for IC*, (College Station, TX), pp. 35–41, Nov. 1989. Private communication of the SRC.

[17] E. Sachs, R.-S. Guo, S. Ha, and A. K. Hu, "Process control system for VLSI fabrication," in *Proceedings to the Fall 1989 Electrochemical Society Meeting*, (Hollywood, Florida), Oct. 1989.

[18] K.-K. Lin, "An expert system for polysilicon recipe generation," Master's Thesis, University of California, Berkeley, July 1987.

[19] D. S. Boning, M. B. McIlrath, P. Penfield, Jr., and E. M. Sachs, "A general semiconductor process modeling framework." to be submitted to *IEEE Trans. Semi. Manuf.*.

[20] H. L. Ossher and B. K. Reid, "FABLE: A programming language solution to IC process automation problems," Tech. Report 248, Computer Systems Lab., Stanford University, 1985.

[21] H. L. Ossher and B. K. Reid, "Manufacturing specification," in *Proceedings of the Second Annual IC Assembly Automation Conference*, (INTEM), Jan. 1986.

[22] C. B. Williams, "Design and Implementation of the Berkeley Process-Flow Language Interpreter," Master's Thesis, UC Berkeley, Nov. 1988.

[23] R. A. Hughes and J. D. Shott, "The future of automation for high-volume wafer fabrication and ASIC manufacturing," *Proc. IEEE*, vol. 74, no. 12, pp. 1775–1793, Dec. 1986.

[24] C. P. Ho, J. D. Plummer, S. E. Hansen, and R. W. Dutton, "VLSI process modeling — SUPREM-III," *IEEE Trans. Electron Devices*, vol. ED-30, no. 11, pp. 1438–1452, Nov. 1983.

[25] M. E. Law and R. W. Dutton, "Verification of analytic point defect models using SUPREM-IV," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, no. 2, pp. 191–204, Feb. 1988.

[26] B. E. Deal and A. S. Grove, "General relationship for the thermal oxidation of silicon," *J. Appl. Phys.*, vol. 36, pp. 3770–3778, 1965.

[27] W. G. Oldham, S. N. Nandgaonkar, A. R. Neureuther, and M. O'Toole, "A general simulator for VLSI lithography and etching processes: Part I – application to projection lithography," *IEEE Trans. Electron Devices*, vol. ED-26, no. 4, pp. 717–722, Apr. 1979.

[28] W. G. Oldham, A. R. Neureuther, C. Sung, J. L. Reynolds, and S. N. Nandgaonkar, "A general simulator for VLSI lithography and etching processes: Part II – application to deposition and etching," *IEEE Trans. Electron Devices*, vol. ED-27, no. 8, pp. 1455–1459, Aug. 1980.

[29] L. A. Rowe, C. B. Williams, and C. J. Hegarty, "The design of the Berkeley process-flow language," Tech. Report No. 90/62, Electronics Research Lab., U.C. Berkeley, Aug. 1990.

[30] J. S. Wenstrand, H. Iwai, and R. W. Dutton, "A manufacturing-oriented environment for synthesis of fabrication processes," *IEEE International Conf. on CAD, ICCAD-89*, pp. 376–379, Nov. 1989.

[31] P. Saha, "IC process synthesis by analytical models," Bachelor's Thesis, Massachusetts Institute of Technology, May 1989.

[32] D. Akkuş, "Process Advisors: Process synthesis for arbitrary initial conditions by analytical models," Bachelor's Thesis, Massachusetts Institute of Technology, May 1990.

[33] C.-Y. Fu, N. H. Chang, and K.-K. Lin, "'Smart' integrated circuit processing," *IEEE Trans. Semiconductor Manufacturing*, vol. 2, no. 4, pp. 151–158, Nov. 1989.

[34] D. C. Montgomery, *Statistical Quality Control.* New York: Wiley, 1985.

[35] H. L. Ossher and B. K. Reid, "Fable: a programming-language solution to IC process automation problems," *Proc. of the SIGPLAN 83 Symp. on Programming Language Issues in Software Systems*, vol. 18, no. 6, pp. 137–148, June 1983.

[36] Consilium, Mountain View, CA, *CAM Systems for Smart Shop Floor Control*, 1986.

[37] Promis Systems Corp., Santa Clara, CA, *The PROMIS System: Controlling the Journey to Factory Automation*, 1987.

[38] O. J. Dahl, "Discrete event simulation languages," in *Programming Languages* (F. Genuys, ed.), (New York), Academic Press, 1968.

[39] R. Glassey, "An Overview of BLOCS/M: The Berkeley Library of Objects for Control and Simulation of Manufacturing," in *1989 DARPA-SRC Workshop on Integrated Factory Managements for Integrated Circuits (IFM-IC)*, (College Station, TX), pp. 81–94, Nov. 1989.

[40] D. T. Phillips, G. L. Curry, and B. L. Deuermeyer, "CHIPS: A Coherent and Integrated System for Semiconductor Manufacturing Systems Analysis," in *1990 DARPA-SRC Workshop on CIM for Integrated Circuits*, (U.C. Berkeley, Berkeley, CA), Aug. 1990.

[41] C. Williams (personal communication), 1990.

[42] F. Maseeh, R. M. Harris, and S. D. Senturia, "A CAD architecture for microelectromechanical systems," in *Proceedings IEEE Micro Electro Mechanical Systems*, (Napa Valley, CA), pp. 44–49, 1990.

[43] F. Maseeh, R. M. Harris, D. S. Boning, M. L. Heytens, S. A. Gelston, and S. D. Senturia, "Application of mechanical-technology CAD to microelectronic device design and manufacturing," in *Proceedings of the International Electronics Manufacturing Technology Symposium*, (Washington, D.C.), Oct. 1990.

[44] M. L. Heytens and M. B. McIlrath, "An Object-Oriented Interface to a Relational Database System," in *1989 DARPA-SRC Workshop on Integrated Factory Managements for Integrated Circuits (IFM-IC)*, (College Station, TX), pp. 165–170, Nov. 1989.

[45] E. Schoen, "HyperClass: Release notes for version 2.0," research note, Schlumberger Palo Alto Research, Apr. 1988.

[46] D. Giuse, "KR: Constraint-Based Knowledge Representation," Memorandum No. CMU-CS-89-186, Carnegie-Mellon, Nov. 1989.

[47] B. M. Leavenworth and J. E. Sammet, "An overview of nonprocedural languages," in *ACM SIGPLAN Symposium on Very High Level Languages*, (Santa Monica, CA), pp. 1–12, Mar. 1974.

[48] H. L. Ossher, "Grids: A new program structuring mechanism based on layered graphs," in *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, (Salt Lake City, Utah), pp. 11–22, Jan. 1984.

[49] B. Liskov, "Abstraction mechanisms in CLU," *Communications of the ACM*, vol. 20, pp. 564–576, 1977.

[50] R. Zippel, "Capsules," *Proc. of the SIGPLAN 83 Symp. on Programming Language Issues in Software Systems*, vol. 18, no. 6, pp. 166–169, June 1983.

[51] M. P. Georgeff and A. L. Lansky, "Procedural knowledge," *Proc. IEEE*, vol. 74, no. 10, pp. 1382–1398, Oct. 1986.

[52] D. S. Boning, M. L. Heytens, and A. S. Wong, "The intertool profile interchange format: An oject-oriented approach." to appear in *IEEE Trans. on Computer-Aided Design*, Sept. 1991.

[53] S. G. Duvall, "An interchange format for process and device simulation," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, no. 7, pp. 741–754, July 1988.

[54] C. H. Corbex, A. F. Gerodolle, S. P. Martin, and A. R. Poncet, "Data structuring for process and device simulations," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, no. 4, pp. 489–500, Apr. 1988.

[55] R. J. Skokel and D. B. MacMillen, "Practical integration of process, device, and circuit simulation," *IEEE Trans. Electron Devices*, vol. ED-32, no. 10, pp. 2110–2116, Oct. 1985.

[56] D. S. Boning and T.-L. Tung, "A Proposal for a Profile Interchange Format Part I: Syntax, Part II: Semantics," VLSI Memo No. 86–356, MIT, Dec. 1986.

[57] K. Kato, N. Shigyo, T. Wada, S. Onga, M. Konaka, and K. Taniguchi, "A supervised simulation system for process and device designs based on a geometrical data interface," *IEEE Trans. Electron Devices*, vol. ED-34, no. 10, pp. 2049–2058, Oct. 1987.

[58] M. Sugimoto and M. Fukuma, "Standard description form for device characteristics in VLSI's," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, no. 2, pp. 293–302, Apr. 1986.

[59] E. G. Schlechtendahl, ed., *Specification of a CAD*I Neutral File for CAD Geometry.* New York: Springer-Verlag, third ed., 1988.

[60] Electronic Design Interchange Format Steering Committee, *EDIF Specification — Version 1 1 0*, Nov. 1985.

[61] T.-L. Tung, "SNC: An Interchange Format for Simulation Programs." Personal communication and software, July 1985.

[62] G. Chin, "User Interfaces for 2D and 3D Simulation," in *August 3, 1988 Research Summary – Process and Device Modeling*, (Stanford University), pp. 73–85, 1988.

[63] A. S. Wong and A. R. Neureuther, "The intertool profile interchange format: A technology CAD environment approach," to appear in *IEEE Trans. Computer-Aided Design*, Sept. 1991.

[64] D. S. Harrison, P. Moore, R. L. Spickelmier, and A. R. Newton, "Data management and graphics editing in the Berkeley design environment," *IEEE International Conf. on CAD, ICCAD-86*, pp. 20–24, Nov. 1986.

[65] J. R. F. McMacken and S. G. Chamberlain, "CHORD: A modulear semiconductor device simulation development tool incorporating external network models," *IEEE Trans. Computer-Aided Design*, vol. CAD-8, no. 8, pp. 826–836, Aug. 1989.

[66] A. Wong and W. Dietrich, "Semiconductor Wafer Representation Architecture, CFI Document No. 162." SWR Working Group of the CFI/TCAD Framework Group, Dec. 1990.

[67] ISO TC184/SC4/WG1, *Information Modeling Language Express – Language Reference Manual*, Oct. 1989.

[68] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon, *Common Lisp Object System Specification.* X3J13 Committee, June 1988.

[69] M. L. Heytens and R. S. Nikhil, "GESTALT: An expressive database programming system," *ACM SIGMOD Record*, vol. 18, no. 1, pp. 54–67, Mar. 1989.

[70] M. E. Law, C. S. Rafferty, and R. W. Dutton, "SUPREM-IV Users Manual," tech. rep., Stanford Electronics Laboratories, Stanford University, Dec. 1988.

[71] R. B. Duncan, "Implementation of graphical analysis and parsing utilities for an IC profile and device structure interchange format," Bachelor's Thesis, Massachusetts Institute of Technology, May 1986.

[72] R. M. Harris and D. S. Boning, "GIRAPHE V3.3: A User's Manual with Examples," VLSI Memo No. 88–486, MIT, Nov. 1988.

[73] H. Afsarmanesh, E. Brotoatmodjo, K. J. Byeon, and A. C. Parker, "The EVE VLSI information management environment," *IEEE International Conf. on CAD, ICCAD-89*, pp. 384–387, Nov. 1989.

[74] A. S. Wong, D. S. Boning, M. L. Heytens, and A. R. Neureuther, "The Intertool Profile Interchange Format," in *Technical Digest of the Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits: NUPAD-III*, (Honolulu, HA), pp. 61–62, June 1990.

[75] A. S. Wong, "An integrated graphical environment for operating IC process simulators," Memo No. UCB/ERL M89/67, Electronics Research Laboratory, U. C. Berkeley, May 1989.

[76] T. Yuasa and M. Hagiya, "Kyoto common lisp report," Technical Report, Research Institute for Mathematical Sciences, Kyoto University, 1985.

[77] G. L. Steele Jr., *Common LISP: The Language.* Burlington, Massachusetts: Digital Press, 1984.

[78] P. Kager and A. Strojwas, "PI/C: Process interpreter/compiler," *IEEE International Conf. on CAD, ICCAD-85*, pp. 321–323, Nov. 1985.

[79] S. R. Nassif, A. J. Strojwas, and S. W. Director, "FABRICS II: A statistically based IC fabrication process simulator," *IEEE Trans. Computer-Aided Design*, vol. CAD-3, no. 1, pp. 40–46, Jan. 1984.

[80] G. C. Clark and R. E. Zippel, "SCHEMA — an architecture for knowledge based CAD," *IEEE International Conf. on CAD, ICCAD-85*, pp. 50–52, Nov. 1985.

[81] H. C. Wu, A. S. Wong, Y. L. Koh, E. W. Scheckler, and A. R. Neureuther, "SIMPL-2 (SIMulated Profiles from the Layout) — Design Interface in X (SIMPL-DIX)," in *Proceedings IEDM 88*, pp. 328–331, 1988.

[82] K. Lee and A. R. Neureuther, "SIMPL-2 (SIMulated Profiles from the Layout - version 2)," in *1985 Symposium on VLSI Technology*, (Kobe, Japan), pp. 64–65, May 1985.

[83] B. R. Penumalli, "A comprehensive two-dimensional VLSI process simulation program, BICEPS," *IEEE Trans. Electron Devices*, vol. ED-30, no. 9, pp. 986–992, Sept. 1983.

[84] D. A. Antoniadis, "Chapter 6: Silicon Integrated Circuit Process Modeling," in *VLSI Electronics, Microstructure Science, Vol. 12, Silicon Materials* (N. G. Einspruch, ed.), pp. 271–306, Orlando: Academic Press, 1985.

[85] S. K. Ghandhi, *VLSI Fabrication Principles.* New York: Wiley, 1983.

[86] G. Freeman, Y.-C. Pan, and W. Lukaszek, "Applicaton of analytic device models to automated IC process diagnosis," in *1990 DARPA-SRC Workshop on CIM for Integrated Circuits*, (U.C. Berkeley, Berkeley, CA), Aug. 1990.

[87] D. S. Boning, "An Opset Generator for the PFR." Internal memo of the MIT CAF project, Aug. 1990.

[88] D. S. Boning, "A Traveler Generator for the PFR." Internal memo of the MIT CAF project, Aug. 1990.

[89] A. J. MacDonald, A. J. Walton, J. M. Robertson, and R. J. Holwill, "Integrating CAM and process simulation to enhance on-line analysis and control of IC fabrication," *IEEE Trans. Semiconductor Manufacturing*, vol. 3, no. 2, pp. 72–79, May 1990.

[90] A. Z. Aktas, *Structured Analysis and Design of Information Systems.* Englewood Cliffs, N.J.: Prentice-Hall, 1987.

[91] J. Mar, K. Bhargavan, S. Duvall, R. Firestone, D. Lucey, S. Nandgaonkar, S. Wu, K.-S. YU, and F. Zarbakhsh, "EASE – An application-based CAD system for process design," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 6, pp. 1032–1038, Nov. 1987.

[92] A. R. Alvarez, B. L. Abdi, D. L. Young, H. D. Weed, J. Teplik, and E. R. Herald, "Application of statistical design and response surface methods to computer-aided VLSI device design," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, no. 2, pp. 272–288, Feb. 1988.

[93] V. Marash and R. W. Dutton, "Methodology for submicron device model development," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, no. 2, pp. 299–306, Feb. 1988.

[94] P. K. Tedrow and C. G. Sodini, "Twin Well CMOS Process, Version 1.2." MIT Microsystems Technology Laboratories, Cambridge, MA, June 1989.

[95] S. Selberherr, A. Schutz, and H. W. Potzl, "MINIMOS – A two-dimensional MOS transistor analyzer," *IEEE Trans. Electron Devices*, vol. ED-27, no. 8, pp. 1540–1550, Aug. 1980.

[96] T. L. Tewksbury III, "Characterization of Nonidealities in Integrated Circuit Transistors and Capacitors," Master's Thesis, Massachusetts Institute of Technology, Sept. 1987.

[97] J. L. McCreary, "Matching properties, and voltage and temperature dependence of MOS capacitors," *IEEE Journal of Solid State Circuits*, vol. SC-16, no. 6, pp. 608–616, Dec. 1981.

[98] Z. Yu and R. W. Dutton, "SEDAN III — A Generalized Electronic Material Device Analysis Program," tech. rep., Integrated Circuits Laboratory, Stanford University, July 1985.

[99] G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters.* New York: Wiley, 1978.

Appendix A

Guide to the Process Flow Representation

Guide to the Process Flow Representation

Version 3.0

Duane S. Boning and Michael B. McIlrath

January 4, 1991

A.1 Introduction

This document describes the basic constructs in the textual form of the MIT Process Flow Representation (PFR). It should be noted that the PFR is currently under development, and the forms described here will change as the PFR evolves. As it does so, new documentation describing the language will be issued.

The written form of the PFR is based on a LISP- or SCHEME-like syntax. In the text to follow, the following symbols are used to summarize the syntax of PFR constructs:

`<value>` A value to be supplied by the writer of a PFR process is indicated by angled brackets.

`[optional]` An optional part of the construct is enclosed in square brackets.

`{repeatable}` A form that may be repeated one or more times is enclosed in braces.

`<option1 | option2>` One of **option1** or **option2** is used (a choice is required, unless enclosed in square brackets).

A.2 Definitions

The `define` construct allows one to associate names with constants or functions.

A.2.1 Constant Definitions

The "constant" use of the *define* is:

```
(define <name> <form>)
```

name The name of the constant being defined.

form The form which is assigned to the constant. The form is only evaluated when the name is actually used.

Constant values may be simple values, such as strings, or complicated forms, such as operations. Examples are:

```
(define GateOxTube "tubeA1")

(define gate-oxide
  (operation
    (:change-wafer-state
      (:deposit :material :oxide
                :thickness (:microns 1)))))
```

It is important to note that the **<form>** is not evaluated at the time of the define, but only when some interpreter invokes the flow evaluator on the name of the definition. In this sense, the **define** acts more like a macro than an assignment (more like a simple **defmacro** than a **setq**). The advantage of this approach is that order of definitions is unimportant (so long as definitions do not conflict or depend cyclicly on one another).

A.2.2 Function Definitions

The **define** construct is also used to parameterize a definition (that is, to define a function). In this case, the form is:

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

name The name of the function.

parameter-name The name of a parameter of the function. The function can take any number of parameters.

parameter-default One can specify default values for the parameters to take if they are not explicitly specified during a function call. If a parameter default is not specified, the default is taken to be **nil**.

forms The forms in the body of the function are evaluated during a function call. The value of the last form is returned by the function call. All appearances of the function parameters in the forms are replaced by the actual arguments of the function call.

A typical use of a function is to parameterize an operation or some commonly used portion of an operation attribute. The following examples illustrate:

```
;; Defining a parameterized operation...
(define (HMDS-prime (recipe 1))
  (operation
   (:machine "HMDS")
   (:settings :material "HMDS"  :recipe recipe)
   (:time-required (:minutes 45))))

;; Defining a parameterized :treatment form...
(define (furnace-dryox-treatment temper-val time-val)
  (:thermal :temperature temper-val
            :time time-val :ambient :02))
```

A.2.3 Function Invocation

A function is called or "invoked" using the following form:

```
(<function-name> {[<argument-keyword> <argument-value>]})
```

function-name The name of the function as specified in the definition.

argument-keyword For each formal parameter specified in the definition of the function, there corresponds a keyworded argument specifier for use in the invocation. This allows one to supply arguments in any order in the function invocation.

argument-value The value of the argument specified by **argument-keyword**. This value is substituted in the function body whenever the formal parameter is encountered.

A word of caution about *constant* versus *function* invocations. Functions must always be enclosed in parentheses, while constant "invocation" (or constant substitution) must *not* be enclosed in parentheses. Confusion is possible, then, when one is using both constant and function definitions. The following three definitions, and their invocations, illustrate the possibilities:

```
;; A constant operation...
(define example1
  (operation ...))

;; A parameterized operation...
(define (example2 temperature (time (:minutes 10)))
  (operation ...))

;; A function that takes no arguments
```

```
(define (example3)
  (operation ...))

;; Invocation examples:
(define calling-operation
  (operation
    (:body
      example1                      ;constant invocation
      (example2 :temperature 900)   ;function with arg
      (example2)                    ;function using defaults
      (example3)                    ;function with no args
      ...)))
```

A.3 Operations

An operation consists of several *attributes*, each one saying something about the operation. Currently a *flow* is identical to an *operation*; the two terms are interchangeable. The basic form for an operation is:

```
(operation
 [(:doc <documentation-string>)]
 [(:version <version-entries>)]
 [(:permissible-delay <delay>)]
 [(: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>)]
 [(:stranger <strange-forms>)])
```

documentation-string A string documenting the operation.

version-entries A sequence of version entries.

delay The amount of time allowed between each operation in the body. This may be one of the keywords **:minimal** or **:infinite**, or may be a specific time. This time applies between each "suboperation" in the body, but says nothing about the time between the current operation and those that might appear before or after it in a higher-level flow.

time-required This describes the time required to complete the entire operation.

body A sequence of component *suboperations* (flows or operations) for this operation.

change-wafer-state A sequence of change-wafer-state primitives.

treatment A sequence of treatment primitives.

machine A string-valued name of a machine.

instructions A sequence of string instructions to issue to the operator during fabrication.

readings A sequence of readings.

settings A sequence of machine settings.

opset-name This is an attribute specific to the MIT MTL. If and only if the operation is an "opset", the name of the opset is specified as a string value.

strange-forms Undefined constructs for experimental extensions.

A.4 Sequences

There are times when one wants to break a description down into smaller parts, but the smaller parts are not bona-fide operations in themselves. The most frequent example of this is in specifying what is happening *within* one of the attributes of a single operation, such as the settings, and one wants to use function definitions to increase modularity in the description. Then a sequence is the correct construct to use, usually in association with *define*.

`(sequence [<form>])`

form Members of the sequence. Valid forms include constants, functions, and inline descriptions of the allowable types, which depend on how and where the sequence is used. For instance, only `:treatment` primitives may be included in a sequence if the sequence is used within a `:treatment` attribute of an operation.

An example of this is shown below:

```
(define (furnace-rampup-treatment final-temperature)
  (sequence
    ;;  Includes push-in, stabilization, thermal ramp from 800 to
    ;; the peak temperature (the parameter), and a stabilization at
    ;; that peak temperature.
    (:thermal :temperature 800
```

```
            :time (:minutes 20) :ambient :N2) ;Push-In
  (:thermal :temperature 800
            :time (:minutes 10) :ambient :N2) ;Stabilization
  (:thermal :temperature 800 :ambient :N2
            :time (:minutes (/ (- final-temperature 800) 10.0))
            :temp-rate 10)              ;Ramp-Up
  (:thermal :temperature final-temperature
            :time (:minutes 10) :ambient :N2))) ;Stabilization
```

The furnace-rampup-treatment function could then be called inside the :treatment part of more than one operation, each time with a different value for the final-temperature. For example:

```
(operation
 (:doc "These two steps have to be done right after each other")
 (:permissible-delay :minimal)
 (:body
  rca-clean
  (operation
   (:doc "SRO furnace processing")
   (:change-wafer-state
    (:oxidation :thickness (:angstroms (:mean 430 :range 20))))
   (:treatment
    (furnace-rampup-treatment :final-temperature (:mean 950 :range 10))
    (furnace-dryox-treatment :temperature 950 :time (:minutes 100))
    (furnace-rampdown-treatment :start-temperature 950))
   (:time-required (:hours 5 :minutes 0))
   (:machine GateOxTube)
   (:settings :recipe 210))))
```

A.5 Version Primitives

There is currently only one :version primitive:

```
(:modified :number <number>
           :by <name>
           :date <date-string>
           :what <description>)
```

number Version number of the change.

name A string giving the name of the person responsible for the change.

date A string giving the date of the change.

what A string description of the change.

A.6 Change-Wafer-State Primitives

The following constructs are currently recognized as primitives of the change-wafer-state attribute of an operation. Within a single change-wafer-state attribute, it is assumed that multiple primitives are all satisfied simultaneously (as opposed to sequentially).

A.6.1 :oxidation

The `:oxidation` primitive specifies the amount of oxide to be grown in some specific area of the wafer.

```
(:oxidation [:thickness <thickness>]
            [:location <cross-section>])
```

thickness The thickness to be grown in the specified cross section. If no cross section is specified, the oxide is grown on what is assumed to be a "virgin", lightly doped, silicon wafer. The amount of oxide grown on other materials (such as polysilicon), or in a region where some oxide already exists, is undefined.

cross-section Where the oxide is to be grown. If specified, then the change in the wafer is such that additional oxide of the specified amount is grown on whatever material is already on the wafer in the specified cross section.

Examples of the use of this primitive are:

```
(:change-wafer-state
  (:oxidation :thickness (:angstroms 430)))

(:change-wafer-state
  (:oxidation :location pwell
              :thickness (:angstroms (:mean 6800 :range 340)))
  (:oxidation :location nwell
              :thickness (:angstroms (:mean 8800 :range 440))))
```

A.6.2 :sinter

The `:sinter` primitive indicates that the wafer has been changed by a low-temperature sinter, that is, good metal-semiconductor contact has been achieved. It takes no arguments.

```
(:sinter)
```

A.6.3 :deposit

The :deposit primitive indicates that material is added to the surface of the wafer. It is assumed that the material is added conformally over the entire surface of the wafer.

```
(:deposit [:material <material>]
          [:thickness <thickness>]
          [:coverage <coverage>])
```

material The material to be added to the wafer.

thickness The amount of the material to be added.

coverage Nature of the deposition step coverage (in the lateral dimensions). Possible values are :flow (planarizing, with thickness defining the thinnest deposition region), :vertical, and :conformal.

An example of the use of this primitive is:

```
(:change-wafer-state
 (:deposit :material :nitride
           :thickness (:angstroms (:mean 1500 :range 150))))
```

A.6.4 :dope

The :dope primitive indicates that the surface of the wafer (assumed to be in a location where silicon is exposed) is to be doped, usually by ion-implantation, but also potentially by solid-state diffusion.

```
(:dope [:dopant <dopant>]
       [:sheet-rho <sheet-rho>])
```

dopant The dopant impurity.

sheet-rho The sheet resistivity of the resulting doped region.

An example of use:

```
(:dope :dopant :P :sheet-rho (:mean 8 :range 2))
```

A.6.5 :implant

The :implant indicates that a specified amount of dopant has been introduced somewhere near the surface of the wafer. The form of the primitive is:

```
(:implant [:element <element>]
          [:dose <dose>]
          [:range <range>]
          [:straggle <straggle>])
```

element The impurity to be introduced into the wafer.

dose The amount of the impurity (in atoms/cm^2) implanted.

range The projected range of the implanted impurity distribution.

straggle The straggle (standard deviation) of the implanted impurity distribution.

Note that the change-wafer-state and treatment :implant primitives are closely related, but only the treatment primitive contains the energy specification. An example showing both the change-wafer-state and treatment primitives for an implant is:

```
(operation
  (:change-wafer-state
    (:implant :element :P :dose 2e12
              :range (:microns .2472) :straggle (:microns .0807)))
  (:treatment
    (:implant :element :P :dose 2e12 :energy 180)))
```

A.6.6 :bake

The :bake primitive indicates that the wafer has been changed by a low-temperature bake such that the indicated organic material (such as resist or polyimide) on the wafer has been hardened.

```
(:bake [:material <material>])
```

material The material to be baked.

A.6.7 :expose

The :expose primitive indicates that the wafer has been exposed to light. This primitive takes a :mask definition as its only argument:

```
(:expose  :mask <mask>)
```

mask The exposure mask.

A.6.8 :develop

The :develop primitive indicates that the wafer has been developed. Any exposed positive resist or any unexposed negative resist is removed. This primitive takes no arguments.

```
(:develop)
```

A.6.9 :etch

The :etch primitive indicates that some amount of material at the surface of the wafer is removed. The form of the primitive is:

```
(:etch :material <material>
       :thickness <thickness>
       :directionality <directionality>)
```

material The material to be removed.

thickness The amount of the material to be removed. This may be either a specific thickness, or the keyword :all.

directionality A measure of the directionality of the etch. Values may be :vertical, :isotropic, or a number from 0.0 to 1.0 indicating the ratio of horizontal to vertical etch.

Examples of the use of this primitive are:

```
(:change-wafer-state
  (:etch :material :nitride :thickness (:angstroms 1500)))

(define (oxide-boe-etch time acid)
  (operation
   (:doc "Generic BOE etch operation.  Parameters include the buffered-oxide
   etchant mixture and the etch time.")
   (:change-wafer-state
    (:etch :material :oxide
           :thickness (* time (etch-rate :acid acid))))
   (:machine  "wet-etch")
   (:settings :sink Oxide-Sink :tank 2 :acid acid :time time)
   (:time-required (* time 2))))
```

A.7 Treatment Primitives

Treatment primitives describe the physical environment around a wafer during processing. If more than one primitive appears within a single (:treatment ...) form, they are assumed to describe the environment sequentially in time.

A.7.1 :thermal

The `:thermal` primitive describes a high temperature furnace environment. The form of the primitive is:

```
(:thermal :temperature <temperature>
          :temp-rate <temp-rate>
          :time <time>
          [:dopant <dopant>]
          [:dopant-concentration <dopant-concentration>]
          :ambient <ambient>)
```

temperature The temperature (in C).

temp-rate The rate at which the temperature changes (in C/minute) during the specified time, starting from the specified temperature.

time The extent of the time for which the wafer is exposed to this environment.

dopant This dopant is in the surrounding ambient gas.

dopant-concentration The concentration of the dopant in the surrounding gas. This may be a specific value (in atoms/cm^3) or the keyword `:solid-solubility`.

ambient The ambient surrounding the wafer. This may be one of the keywords `:O2`, `:H2O`, or `:nitrogen`.

A.7.2 :epitaxy

The `:epitaxy` primitive describes a high temperature furnace environment during which epitaxial silicon is grown. The form of the primitive is:

```
(:epitaxy :temperature <temperature>
          :temp-rate <temp-rate>
          :time <time>
          [:dopant <dopant>]
          [:dopant-concentration <dopant-concentration>]
          :growth-rate <growth-rate>)
```

temperature The temperature (in C).

temp-rate The rate at which the temperature changes (in C/minute) during the specified time, starting from the specified temperature.

time The extent of the time for which the wafer is exposed to this environment.

dopant This dopant is in the surrounding ambient gas.

dopant-concentration The concentration of the dopant in the surrounding gas. This may be a specific value (in atoms/cm^3) or the keyword `:solid-solubility`.

growth-rate The rate (in microns/minute) at which the epitaxial layer grows.

A.7.3 :implant

The `:implant` indicates that a specified amount of dopant at a specified energy is implanted into the surface of the wafer. The form of the primitive is:

```
(:implant :element <element>
          :dose <dose>
          :energy <energy>)
```

element The impurity to be introduced into the wafer.

dose The amount of impurity (in atoms/cm^2) implanted.

energy The energy (in KeV) of the implant.

Note that the change-wafer-state primitive `:implant` is closely related, but only the treatment primitive contains the energy specification.

A.8 Machine Construct

The syntax for the `:machine` construct is shown below. In normal use, the `:machine` construct will be accompanied by a `:settings` construct within the same operation to identify the settings that are to be used on the specified machine.

```
(:machine <machine-spec>)
```

machine-spec This is the name of the machine to be used, expressed as a string. This may be the form `(:choose {<machine>})`, where several alternative machines may be listed.

A.9 Settings Construct

Machine settings provide information on how to perform an operation on a particular machine (as specified using the `:machine` construct). An individual setting appears as a pair of keywords and values. If more than one pair appears within a settings (i.e. `(:settings :key1 val1 :key2 val2 ...)` form, they are assumed to be settings that must all be "loaded" or understood by the machine or operator before

the operation begins. If a sequence of grouped settings appear within a single settings (i.e. `(:settings (:key1 val1) (:key2 val2))`) they are assumed to describe sequentially in time settings to be performed.

The PFR does not itself restrict the keywords and values that may be used within the `:settings` construct. Two ways of using these keywords are possible. First, the keyword-value pairs may be understood to be a "specification" not only of the values of the various settings, but also of the particular kinds of settings that are desired to be possible for the operation. The second mode is more common, where the possible keyword-value pairs are defined by the database of individual machines known to the surrounding fabrication system. Because these are site-specific, they are not described in this guide, but are deferred to site-specific documentation. An example use of the `:settings` is shown in the following example:

```
(define junction-drive
  (operation
   (:doc "Set the junction depths to match Building 13 process.
     Does an additional dry02 drive, to match what used to be a
     densification in Building 13".)
   (:treatment
     (furnace-rampup-treatment :final-temperature 950)
     (furnace-dryox-treatment :temperature 950 :time (:minutes 15))
     (furnace-rampdown-treatment :start-temperature 950))
   (:machine (:choose ThickOxTube ThickOxTube2))
   (:settings :recipe 602)))
```

A.10 Readings Construct

The `:readings` construct specifies data collection requirements for an operation. Usually this is to record some piece of data from the machine or measurement equipment. In some cases, this data collection might be automated, while in others the user may be required to enter the data. The PFR does not, however, make any assumptions about the mechanism for the collection of this data. The form of the `:readings` construct is:

`(:readings {<reading-pair>} | {<readings-sequence>})`

reading-pair A reading pair has the form `(<what> <description>)`, and specifies an individual reading to be taken.

what A keyworded description of the kind of reading that is to be taken. This keyword might be expected to be understood by the measurement and readings subsystem of the fabrication system, so that certain kinds of reasoning on the results are possible. Currently, this set of readings includes `:current`, `:time`,

`:focus`, `:exposure`, `:alignment`, `:etchtime`, `:thickness`, `:sheet-rho`, and `:refractive-index`.

description A text string describing (or prompting for) the reading value.

readings-sequence More than one `<reading-pair>` (or additional readings-sequences) can be specified enclosed in a sequence. It is currently assumed that requests for such readings will be issued sequentially, but that filling these readings may occur in any order.

Examples illustrating this construct are:

```
(define (resist-inspect (where ""))
  (operation
   (:time-required (:minutes 30))
   (:instructions
     (|| "Inspect Wafer Alignment (Flat Away) "
      where " Spec +/-.5u:"))
   (:readings
     (:alignment "Left side x")
     (:alignment "Left side y")
     (:alignment "Right side x")
     (:alignment "Right side y"))))
```

A.11 Instructions Construct

The `:instructions` construct provides for a textual instruction to be issued to an operator.

```
(:instruction {<instruction-string>} | {<instruction-sequence>})
```

instruction-string A string containing the instruction to the operator. It is often the case that the `flow-string` function will be used to build a string given some number of arguments.

instruction-sequence More than one `<instruction-string>` (or additional instruction sequences) can be enclosed in a sequence form. It is assumed that all instructions within a surrounding `:sequence` construct will be issued in order, and all without intervening pause or user response.

Examples illustrating this primitive are:

```
(:instructions "Visual All Wafers")

(define (inspect-instructions film-type (spec :unknown)
                              (what "") (where "wafer"))
  (sequence
   (flow-string "Inspection of ~A" what)
   (flow-string "Film: ~A" film-type)
   (flow-string "Spec: ~A" spec)
   (flow-string "Where: ~A" where)))

(:instructions (inspect-instructions :what :thickness
                                     :film-type :oxide
                                     :spec (:angstroms 120)
                                     :where "Center Wafer"))
```

A.12 Stranger Constructs

The :stranger construct provides a mechanisms for the site- or application-specific definition of extensions to the PFR. Only the attributes listed in Section A.3 can be associated with an operation (so that syntactic errors may be detected). Extensions to the PFR that are not "standard" or mutually agreed upon can be included in the :stranger attribute. The general form is

```
(:stranger [{(<application-key> <application-forms>)}])
```

application-key A keyword identifying the succeeding application forms. Multiple sets of such constructs may appear within the same :stranger form.

application-forms The contents of the extension form. The structure of these forms is defined externally.

An example illustrating the :stranger extension is:

```
(:stranger (:opset "phwell.set")
           (:suprem3 "Comment      Suprem-III Model fix"
                     "Phosphorus   dix.0=2.31E12''))
```

A.12.1 Suprem3-code Stranger Construct

The :suprem3-code stranger construct provides a mechanism to override the code that is generated automatically by a PFR to Suprem-III translator.

```
(:suprem3-code {<statement-string>})
```

statement-string A Suprem-III statement, or sequence of statements.

An example illustrating this construct is:

```
(define boron-model-quick-fix
  (flow
   (:stranger
    (:suprem3-code
      "Comment     -- A fake diffusion to fix lack of a"
      "Comment        kinetic boron model in Suprem-III"
      "Diffusion   Temp=950 Time=.0001"))))
```

A.12.2 simpl-code Stranger Construct

The `:simpl-code` stranger construct provides a way to override the code that is generated automatically by a PFR to SIMPL2 translator.

```
(:simpl-code
 [{(:depo :material <material>  :thickness <thickness>
          [:iso <iso:"V">]      [:angle <angle:45>])}]
 [{:devl  :layer <layer>)}]
 [{(:etch :layer <layer>)}]
 [{(:expo :mask <mask>          [:invert nil]
  :material <material>  :exposed-material <expo-material>)}]
 [{(:impl :impurity <impurity>  :dose <dose>
  :std-dev <std-dev>    :depth <depth>
          [:block-thick <block-thick:depth+3*std-dev>])}]
 [{(:oxid :thickness <thickness>)}]
 [{(:save <filename>)}])
```

The arguments to these forms are directly derived from the SIMPL-2 manual, and that manual should be consulted for their explanation. The `:simpl-code` stranger forms can be used, along with the function definitions in the PFR, to directly code SIMPL-2 programs in a more convenient and flexible form than that provided by SIMPL-2 itself. All of these forms are not necessary, as the change-wafer-state descriptions of the PFR can be used instead by the PFR to SIMPL-2 translator to generate the necessary input to SIMPL-2.

A.13 Inexact Numbers

There are forms in the PFR for describing simple distributions of numeric parameter values. The possible forms are:

```
(:mean <mean-value> [:range <range-value>])
(:mean <mean-value> [:plus-range <plus-range>]
                    [:minus-range <minus-range>])
```

mean-value A form expressing the mean value.

range-value A form expressing the range of the distribution. Using a single range is equivalent to specifying both the **plus-range** and the **minus-range** to be this value.

plus-range The positive deviation of the values from the mean. The upper limit to the values is **mean-value** + **plus-range**.

minus-range The negative deviation of the values from the mean. The lower limit is **mean-value** − **minus-range**.

Note that the forms used for mean and range information may have units associated with them, or the inexact value may be used within a unit specifying form, as in the examples below:

```
(:mean (:microns 1) :range (:angstroms 100))

(:angstroms (:mean 100 :plus-range 10 :minus-range 20))
```

A.14 Units

The PFR understands a limited set of forms that specify units information along with numeric values. In each case, rather than specifying a pure numerical value, one can use a form such as:

```
(<units> <value>)
```

units The keyworded units value. These are described below.

value The numeric value in the given units.

In each case, the units enable the PFR to convert to the "base" units used by the PFR. One can omit the units altogether and use the numeric value directly if one is very careful to always express the value in the base units of the system; because this base may change, it is recommended that one always supply units information. Furthermore, it is possible to specify a value in mixed units of the same time. For instance, one can specify a time corresponding to 130 minutes as

```
(:hours 2 :minutes 10)
```

The units understood by the PFR are described below.

A.14.1 Length Units

The base unit of length in the PFR is angstroms (`:angstroms`). Other units understood include `:nm` (nanometers) and `:microns`.

A.14.2 Time Duration Units

The base unit of time duration in the PFR is seconds (`:seconds`). Other units of time understood are `:minutes` and `:hours`.

A.14.3 Temperature Units

The base temperature unit is degrees C. There current exists no keyworded temperature unit constructs; you should always use degrees C.

A.15 Arithmetic Expressions

One can perform arithmetic calculations on simple numeric values and inexact numeric values. These expressions are of the form shown below:

`(<operator> {<values>})`

operator The operator may be one of +, -, * (multiply), or / (divide).

values One or more values on which to operate.

A.16 Comparison Expressions

One can perform arithmetic and string comparisons on values. These comparisons are of the form shown below:

`(<predicate> {<forms>})`

predicate The predicate may be one of =?, >?, <?, >=?, <=?, or NOT.

forms One or more values (or expressions) on which to operate.

An example of a comparison expression is:

```
(not (>? 27 30)) => T (true)
```

A.17 String Functions

A.17.1 Concatenation (||)

Multiple strings can be concatenated together into a single string using the concatenation function (||). The form for the use of || is:

```
(|| {<string> | <form>})
```

string This is a simple string. It may have embedded newlines.

form A printable (string) representation of any non-string form will be generated for concatenation.

A.17.2 Flow-String

One can control the construction of a string (more completely than by concatenation) using the `flow-string` function. The function syntax is:

```
(flow-string <format-string> [{<args>}])
```

format-string This is a string with internal formatting descriptions. A subset of the Common Lisp formatting commands may be used. These include include "~A" for a printable version of some object, "~%" for a newline, "~D" for an integer, "~E" for printing an number in exponential style, "~F" for a floating point number, and "~G" for printing a general number.

args These are the arguments that are substituted into the string as mandated by the format-string specification.

A.18 Conditional

The `if` special form provides branching on a condition:

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

condition The condition on which to branch.

then-clause The result if the condition evaluates to non-nil (true).

else-clause The result if the condition evaluates to nil (false).

An example illustrating the use of the conditional is show below:

```
(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
```

A.19 Special Functions

The PFR contains a limited number of "built-in" functions. These are listed below.

A.19.1 elapsed-time

The `elapsed-time` function (which takes no arguments, and is invoked by `(elapsed-time)`) returns the time that has elapsed (in seconds) since the wafer last underwent processing. The function may be extended in the future so that one may specify which operation or type of operation marks the time of interest.

A.19.2 time-string

The `time-string` function takes one argument, a time, and returns a string that expresses that time in human-readable form. This function is useful for formatting strings in the `:instruction` settings primitive.

A.20 :Unknown Value

Often, particularly during the design of a process, some value is not yet known. The PFR allows one to use the special value `:unknown` to indicate this. Having an unknown value is an error for some kinds of interpretations (such as fabrication), while other interpreters may be able to manipulate or make calculations based on the presence of an unknown value.

A.21 Mask Definition

A mask describes a physical glass plate generated from the boolean combination of one or more generic layout layers. A mask is specified using one of the two forms:

```
(:mask         <layers-list> :mask-name <mask-name>)
(:mask-inverse <layers-list> :mask-name <mask-name>)
```

:mask The mask is made by OR'ing together the specified layout layers. This generally corresponds to a clear-field mask.

:mask-inverse The mask is made by first OR'ing together the specified layout layers, and then inverting. This generally corresponds to a dark-field mask.

layout-layers This is a list of strings specifying the layout layers that are OR'ed together to make the mask.

mask-name A name used to refer to the mask.

Often a constant is defined for the mask, and then that constant is used whenever a mask specification is needed in a particular flow. Examples of such a definition are shown below.

```
(define CPF (:mask-inverse ("CW" "CPFV") :mask-name "CPF"))
(define CNF (:mask ("CW" "CNFV")         :mask-name "CNF"))
```

A.22 Cross Section Definitions

So far, the PFR only supports specification of one-dimensional cross sections. These correspond to "drill-holes" at generically located regions on the wafer. Rather than define a specific coordinate on a specific wafer or mask, one defines a cross section by the intersection of a set of layout layers. The basic form is shown below:

```
(:cross-section <layer-list> :section-name <section-name>)
```

layer-list An exhaustive list of the names of the layout layers that are present at this particular cross section.

section-name The name of the cross section.

Examples showing the definition of a couple of constant cross-sections appear below:

```
(define nchan (:cross-section ("CW" "CD" "CP") :section-name "nchan"))
(define pchan (:cross-section ("CS" "CD" "CP") :section-name "pchan"))
```

A.23 Acknowledgments

The Process Flow Representation has been conceptualized, defined, and implemented by members of the MIT CAF project, including Mike McIlrath, Duane Boning, Don Troxel, and Paul Penfield. The PFR has been influenced by the work of groups at UC Berkeley and Stanford University.