THE DESIGN OF THE BERKELEY PROCESS-FLOW LANGUAGE

by

Lawrence A. Rowe, Christopher Williams,
and Christopher Hegarty

Memorandum No. UCB/ERL M90/62

26 July 1990

COVER PAGE

THE DESIGN OF THE BERKELEY PROCESS-FLOW LANGUAGE

by

Lawrence A. Rowe, Christopher Williams,
and Christopher Hegarty

Memorandum No. UCB/ERL M90/62

26 July 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

THE DESIGN OF THE BERKELEY PROCESS-FLOW LANGUAGE

by

Lawrence A. Rowe, Christopher Williams,
and Christopher Hegarty

Memorandum No. UCB/ERL M90/62

26 July 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The Design of the Berkeley Process-Flow Language[†]

(Draft printed: July 25, 1990)

Lawrence A. Rowe, Christopher Williams, and Christopher Hegarty

Computer Science Division - EECS
University of California
Berkeley, CA 94720

Abstract

The design of a process-flow representation is described that can be used to specify the information needed to design, test, and execute processes to manufacture integrated circuits. A programming language representation is used so that a full complement of programming constructs are available (e.g., data structures, control structures, procedural and object-oriented abstractions, and exception handling).

Programs written in the Berkeley Process-Flow Language (BPFL) are read and interpreted by other programs, called interpreters, that perform tasks such as work-in-progress tracking, process simulation, and factory scheduling.

The hardware and software architecture of a distributed computer-integrated manufacturing system, the design of BPFL, and the features provided to support the WIP and simulation input generator interpreters are described.

1. Introduction

A formal process-flow specification is an important component of an integrated circuit (IC) computer-integrated manufacturing (CIM) system. A process-flow specification includes a complete description of the information needed to manufacture an IC (e.g., masks, materials, equipment, operations, and tests). Treated as a program, the specification can be executed by a computer, which will reduce manufacturing problems.

Manufacturing problems occur for many reasons including human errors (e.g., an operator may enter the wrong parameters for a recipe), equipment failures (e.g., valve leaks), material problems (e.g., wafer contamination), and environmental problems (e.g., water contamination). Human errors can be reduced by connecting equipment to the CIM system and minimizing the need for actions by an operator. The remaining problems can be reduced by careful monitoring and checking during processing. The goal is to gain better control of the process. This goal can be achieved by automated processing, which

[†] This research was supported by the National Science Foundation (Grant MIP-8715557) and The Semiconductor Research Corporation, Philips/Signetics Corporation, Harris Corporation, Texas Instruments, National Semiconductor, Intel Corporation, Rockwell International, Motorola Inc., and Siemens Corporation with a matching grant from the State of California's MICRO program.

requires an executable process-flow specification.

A formal process-flow specification can also dynamically change the processing for a lot based on data gathered during prior processing of the lot or during processing of other lots on the equipment. Changing the processing that a lot receives is called *feed-forward* or *feed-backward* control depending on whether the change is influenced by processing on the same lot in the past or lots that are processed before it.

Lastly, a process-flow specification can be used as input to programs other than a shop floor control system to improve other aspects of IC process design and manufacturing (e.g., process simulators, factory scheduling systems, and process checking systems). Each of these programs currently uses different specifications than the one used by the shop floor control system. Consequently, these different specifications are often inconsistent in the sense that they describe different processes. Failure to maintain consistent specifications can significantly reduce productivity. The development of one process-flow specification that can serve many uses can eliminate these problems.

Current process-flow specifications are either structured documentation (e.g., the TI TRAV system [4]) or run-sheet specifications (e.g., PROMIS[1] and WORKSTREAM[2]). These specifications are typically used only for shop floor control. In a structured documentation system, a printed copy of the specification, called a *traveler*, is taped to the lot and passed along with it to the different workcells. The operator follows the instructions on the traveler that describe how the lot should be processed. A run-sheet system is essentially an interactive traveler. It describes the processing at each workcell and where the lot should be moved when the current step is finished. Instructions are displayed to an operator on a terminal. Some run-sheet systems issue commands to execute recipes in equipment that is connected to the system and to direct a material movement system to move a lot to a different workcell [8].

The problem with these systems is that they do not automate fabrication. They track work-in-progress (WIP) and provide much needed production management information but they cannot support equipment integration and feed-forward and feed-backward process control. Run-sheet specification languages are better than structured documentation systems because they automate some operations, but they are typically limited to a small, fixed set of commands. The specification languages provide operator input/output commands, material movement commands, and data collection and archiving commands.[3]

[1] PROMIS is a product of Promis, Inc., Toronto, Canada.

[2] WORKSTREAM, formerly called COMETS, is a product of Consilium, Inc., Mountain View, California.

[3] WORKSTREAM has a more powerful scripting language with some control structures that can be called from a run-sheet command. However, many commands that can be executed in a script cannot be executed directly in the run-sheet, which severely limits flexibility of the system.

These commands are adequate for tracking WIP and eliminating some operator errors. However, they do not support direct communication with equipment, feed-forward and feed-backward control, and diagnosis and correction of exceptional conditions (e.g., detecting equipment that violates qualification standards).

The fundamental problem with these systems is that they do not provide the power and flexibility of a modern programming language. A process-flow specification is essentially a program for a very complex system composed of equipment in the fabrication line, the material movement system, and data stored in databases that describe the factory state and processing history. Moreover, they only specify production processing. They do not include the information needed by process simulators, schedulers, and other programs that operate on process-flows.

Several research groups are working on process-flow representations that have the power of a modern programming language and that include information needed by other programs. Two kinds of representations are being explored by different groups: 1) a knowledge-based representation and 2) a programming language representation. The knowledge-based approach uses a hierarchical, object-oriented data structure to represent a process-flow (e.g., MKS being developed at Schlumberger and Stanford [16,29], STDS being developed at TI [11], and PFR being developed at MIT [12]). The data structure is composed of objects that represent operations. An operation can be an equipment operation, a control operation (e.g., a conditional, looping, or procedure call command), an input/output operation, or a database operation. A class is defined for each operation. The class contains fields that specify the parameters for the operation. A method is defined on the class that defines the semantics of the operation.

The advantage of this representation is that new operations can be defined as a subclass of an existing class so that default parameters of the new operation can be inherited from the existing operation. Other advantages are that the data structure can be conveniently stored in a relational database (e.g., an object is stored as a tuple in a relation) and it is relatively easy to write programs that access and update a process-flow because it is just a data structure.

The disadvantage of the knowledge-based approach is that the knowledge representation system does not include sophisticated control structure abstractions required to handle unexpected situations (e.g., a furnace run aborts because of a power failure or a constraint on the maximum time delay before starting a furnace operation is violated).[4] This representation emphasizes the correct behavior of the process.

The programming language approach uses a procedural representation for a process-flow (e.g., the Berkeley Process-Flow Language, FABLE developed at Stanford [15], and

[4] Anecdotal evidence from other programming applications suggests that as many as 50% of the lines of code deal with unexpected and error situations.

MPL.2 being developed at Siemens [31]). A process-flow is composed of a collection of procedures that contain conventional programming language commands (e.g., variable and data declarations, assignments, control structures, etc.) interspersed with commands to communicate with equipment, operators, and the database. The process-flow is compiled into an abstract syntax tree which is equivalent to the data structure in the knowledge-based representation. The advantage of the procedural representation is that a full complement of control-structures, data structures, and programming notations are available. This representation emphasizes both the correct and incorrect behavior of the process.

The major difference between the two approaches, besides the different aspects of the process they emphasize, is the use of a procedure call with default parameters as opposed to a subclass with inheritance to define new operations in terms of existing operations. Consequently, the kind of representation is less important than the particular constructs and abstractions that are provided.

This paper describes the Berkeley Process-Flow Language (BPFL). The language is designed to allow all information about a process to be merged into a common specification. Different programs, called *interpreters*, read a BPFL program and perform a task (e.g., run or simulate a process). An interpreter is a program that executes a BPFL program. Each command is either a language primitive (e.g., an assignment command), a built-in procedure call (e.g., a procedure that allocates wafers), or a user-defined procedure call (e.g., a procedure that performs a hard-bake operation by loading a furnace and invoking the appropriate equipment recipe). For example, a WIP interpreter executes a BPFL program and in the process issues instructions to operators and equipment to process lots. A simulation input-generator interpreter, on the other hand, executes the program and outputs commands that can be input to a process simulator to produce a wafer profile (e.g., PROSE [32], SIMPL [9] or SUPREM [7]). Other interpreters can produce input for a factory simulator (e.g., MODES [10], CHIPS [17], and BLOCS [5]). or check process correctness (e.g., check that resist is removed before executing a furnace operation).

The design goals for the language are:

(1) to share as much of a process specification as possible between the different interpreter views (e.g., specify common parameters in only one place),

(2) to provide support for a complete specification including lot splits and merges, exception handing, feed-forward and feed-backward control, and equipment and operator communication, and

(3) to separate the facility specific information from the process specification to make it easier to change equipment in a fab or move processes to a different fab.

The remainder of this paper describes the BPFL language and the WIP and simulation input-generator interpreters. It is organized as follows. Section 2 describes the architecture of the CIM system and how the shared database and process-flow specification fit into it. Section 3 describes the global structure of a BPFL process-flow and how to specify different views of a process-flow. Sections 4 and 5 describe the fabrication and simulation views. Section 6 describes the implementation status and our experiences with BPFL. Lastly, section 7 presents conclusions.

2. System Architecture

This section describes the logical architecture of the hardware, software, and databases that use BPFL. We envision the system running on a distributed heterogeneous computing system connected by local area networks. A typical fab uses large microcomputers for cell controllers, a large mini- or mainframe computer for area and factory control, and a collection of workstations and terminals for user interactions. Figure 1 shows a typical system. Notice that cell controllers have local databases and the fab has a large shared database server, which motivates the need for a distributed database. Terminals and workstations are provided where appropriate. They can be connected either to a terminal server or to a convenient computer. Lastly, equipment is connected to the cell controllers.

This architecture suggests a hierarchical structure (i.e., a cell computer is subordinate to an area computer which is subordinate to a factory computer) but in fact programs on any computer can access databases and programs running on any other computer using an

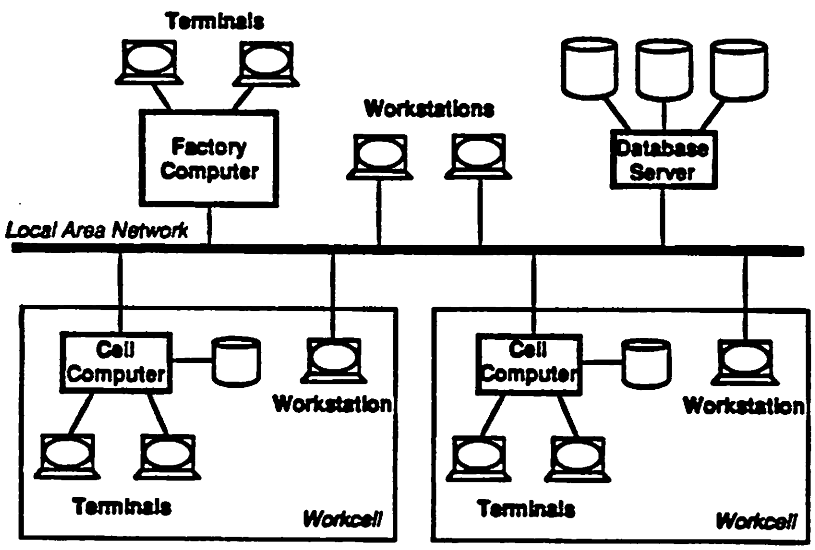


Figure 1: Typical fab computing system.

interprocess communication protocol.

A key component of the system is a shared CIM database that stores all information about the design, manufacture, and sale of semiconductors. This database contains information about the fabrication facility (e.g., rooms and areas, equipment, and utilities), process-flow specifications, WIP (e.g., lots, wafers, data collected during processing, material, etc.), equipment (e.g., status, recipes, qualification and maintenance logs, reservations, etc.), test data, product inventory, and orders. While the database is treated logically as a single centralized database, the architecture that we envision stores data in a distributed heterogeneous database (e.g., GESTALT [2] or INGRES/STAR[5]). Data will be located on the computer that optimizes the cost, reliability, and access constraints imposed by its use. For example, equipment recipes are stored in databases on the cell computers, test data is stored on area computers in testing, and production schedules are stored on the factory computer.

A heterogeneous distributed DBMS is required for two reasons. First, different applications in the fab have different data requirements. One DBMS cannot satisfy all these requirements. For example, the real-time performance and data volume required by some on-line monitoring applications can only be met today by file systems.

Second, it must be possible to integrate existing applications that use older technology storage systems (e.g., VSAM files and IMS databases) with this new architecture. It would be prohibitively expensive to rewrite all corporate applications (e.g., order processing and product inventory) because a new system was introduced into the fab. However, these new applications often must access the data managed by these older corporate applications. A distributed heterogeneous DBMS that provides gateways to different storage systems will allow these fab applications to access new and old data. Consequently, data will be stored in many different systems including third generation database systems (3GDBS) [24], conventional database systems (e.g., relational, network, and hierarchical), and files where appropriate.

A 3GDBS is required for many CIM applications. A 3GDBS supports relational data storage and access, an object-oriented data model (i.e., inheritance, user-defined data types, and methods), and a rules system. An example is the POSTGRES system being developed at Berkeley [23,25]. A 3GDBS can store and access data that cannot be stored and accessed easily in a conventional relational database. For example, measurements collected during wafer processing are often represented by a sequence of values with units. A 3GDBS can store arrays of user-defined data types (e.g., values with unit designations) in a table.

BPFL uses the CIM database in several ways. First, BPFL programs themselves are stored in the database. A software version control system is implemented on top of the

[5] INGRES/STAR is a product of Ingres Corporation, Alameda California.

DBMS to manage libraries of BPFL procedures and their status (e.g., under development, approved for use in a particular fab, etc.).

Second, BPFL interpreters use information in the CIM database. For example, the equipment in the fab and its current status is maintained in the database [13]. The scheduler uses this data to determine which piece of equipment should be allocated to a run. Another example is the WIP system itself, which stores the state of all active runs in the database so that the system can recover from a failure (e.g., a computer crash). Mirrored disks, on-line backup and recovery, and standby spare databases [3] can reduce the possibility that information is lost and reduce the down time should a failure occur.

Third, BPFL programs store and access data in the CIM database. For example, an event log that records the start- and end-times of operations, measurements, and other processing information is stored in the database. The log can be accessed by a BPFL procedure to change future processing based on previously recorded measurements (i.e., feed-forward or feed-backward control). In other words, the database is a convenient repository for data that is shared within a run and between runs.

3. BPFL Program Structure

This section describes the global structure of a BPFL process-flow and the wafer, lot, and view abstractions supported by the language.

The current version of BPFL is implemented as an extension to Common Lisp [22]. Lisp was chosen as the host language for several reasons. First, it is very easy to develop programs that manipulate other programs in Lisp since programs are represented using the native list data structures. The interpreters discussed above operate on BPFL programs so Lisp simplified their development. Second, Lisp provides a very powerful and flexible framework within which to experiment with language designs. This version of BPFL is the second version of the language and it is very different from the first version [30]. We could not have completed this redesign of the language and reimplementation of the interpreters if we were implementing the language from scratch. Lastly, a well-defined and powerful object-oriented programming model, the Common Lisp Object System (CLOS), was already available in the language. CLOS is used extensively both in the design of the language and in the implementation of the interpreters.

The Lisp representation of BPFL programs described in this paper is not the program representation that users will see. A forms-based, graphical user-interface can be implemented that provides a user friendly interface. Examples of such interfaces are the graphical representation of process-flows in the MKS system [16], the experimental user interface to a process-flow representation developed at TI [6], and the Stanford Graphical Design Toolkit [26].

A process-flow is represented by a BPFL procedure that contains a sequence of steps. Each step contains a sequence of function calls on BPFL procedures or primitives or Common Lisp functions. Figure 2 shows the top-level process-flow definition of the standard CMOS process that is run at Berkeley.[6] BPFL procedures are defined by the *defflow* construct. This construct has four arguments: the procedure name, a documentation string, the formal argument list, and a procedure body. The procedure body contains a sequence of process steps. The first step allocates the wafers that will be processed by the run. The second step measures the resistivity of the wafers to insure that they are within tolerances. The third step creates a well. The Berkeley *CMOS-16* process-flow has 12 top-level steps.

The first argument to the *step* construct is a symbol that names the step. It is used to document the step. It is also used in step paths, described below, to specify where certain events occurred in a process-flow.

BPFL provides built-in abstractions to manipulate wafers and lots. Wafers are represented by CLOS objects. Each wafer is assigned a globally unique number that distinguishes it from all other wafers. This number corresponds to the identifying code that is scribed onto the back of a wafer. The wafer is also assigned a logical wafer number within the run. For example, the *i*-th wafer allocated within a run is assigned logical number *i*. These logical wafer numbers are used to reference the wafer within a run. Wafer objects also have properties that specify the resistivity, orientation, and type of the wafer.

A *lot* is a named set of wafers. Predefined lot names are supplied for wafers that will be used to produce product (`product`), wafers that should be scrapped (`scrap`), and wafers that need rework (`rework`). There is also a current lot that contains the wafers

```
defflow CMOS-16(masks, lot-size)
  "U.C. Berkeley Generic CMOS Process (Ver 1.3 14-April-89)
   (2 um, N-well, single poly-Si, single metal)"
begin
   step ALLOCATE-WAFERS: ...;
   step MEASURE-RESISTIVITY: ...;
   step WELL-FORMATION: ...;
   ...
end;
```

Figure 2: Berkeley CMOS-16 process-flow.

[6] A modified notation similar to a conventional algebraic language is used instead of Lisp to make the programs easier to read. Keywords are displayed in a bold font. Identifiers and constants are displayed in a roman font. Keywords and identifiers may contain the dash character ('–') to improve their readability.

that an operation should process (current). A BPFL program can define new lots to hold test wafers or to identify subsets of the product lot that will receive special processing. For example, a process might perform measurements on special test wafers. A wafer can be in more than one lot at the same time. During processing for example, a wafer might need rework in which case it will be in both the product lot and the rework lot.

Functions are provided to access all wafers allocated during a run, all wafers in a lot, the names of lots created during a run, and the lots that contain a particular wafer. In addition, functions are provided to create and destroy lots and add wafers to and remove wafers from lots. A lot split operation can be represented either by creating a new lot and dividing the wafers between it and the previously existing lot or by spawning a new run and passing it a set of wafers. The first representation is appropriate when special treatments within a process are required and the second representation is appropriate when a run is being split into separate runs to improve the degree of concurrent processing. Functions are also provided to merge lots.

Figure 3 shows the wafer allocation step in the Berkeley CMOS process. Arguments can be passed to procedures either by position or by name. Arguments passed by name can be passed in any order because the formal argument name precedes the value in the call. For example, the *bare-silicon-wafer* procedure in the figure uses named argument passing to pass its three arguments: the desired crystal orientation of the wafers (*crystal-size*), the desired resistivity (*resistivity*), and the desired dopant (*dope*).

```
step ALLOCATE-WAFERS: begin
    /* Allocate a device lot and 2 test wafers */
    let spec := bare-silicon-wafer(crystal-face: '<100>,
                resistivity: [{18 ohm-cm},{22 ohm-cm}], dope: 'p);
    begin
      allocate-lot(size: lot-size, snapshot: spec, lot-name: 'product);
      allocate-lot(size: 1, snapshot: spec, lot-name: 'well);
      allocate-lot(size: 1, snapshot: spec, lot-name: 'nch);
    end;
  /* divide product into sublots for different ion implants */
  lot-split(lot: 'product into: '(high medium low));
  set-current-lot('product);
end;
```

Figure 3: Wafer allocation step.

The first operation creates a specification for the types of the wafers to be allocated. Notice that the *resistivity* argument is enclosed in square brackets. This notation specifies a range constant which in this case is a value between 18 and 22 ohm-centimeters. The values in the range constant are enclosed in set brackets. This notation specifies a value with unit designation. The first expression is the magnitude and the second expression is the unit designation. For example, *{18 ohm-cm}* denotes the value 18 ohm-centimeters.

The next three operations allocate wafers and assigns them to lots. The *size* argument specifies how many wafers to allocate, the *snapshot* argument specifies the required properties for the wafers, and the *lot-name* argument specifies the lot in which to put the wafers. The first *allocate-lot* call allocates the product wafers and the next two calls allocate test wafers.

After the wafers are allocated, the product lot is divided into three subsets that will receive different treatments at the ion implanter. The *lot-split* function partitions wafers from one lot into one lot to one or more other lots. It does not remove wafers from the source lot, which in this case is the `product` lot. The last operation in the step sets the `current` lot to the `product` lot, which contains the wafers on which product will be fabricated.

Figure 4 shows the *WELL-FORMATION* step. This step contains five operations. The first operation creates a sacrificial oxide using the *wet-oxide* BPFL procedure. The lots to which the operation should be applied are specified in the *with-lot* statement. Notice that the `product` lot and `nwell` test lot are processed by this operation. The second operation calls the *pattern* procedure, which performs a lithography operation. The argument specifies the mask to use in the exposure operation, which in this case is the *NWELL* mask.

The third operation calls the *implantation* procedure, which performs an implant operation. The arguments specify the dopant, dose, and energy level for the implant and a parameter for a SIMPL interpreter.

The fourth operation is a nested step that drives-in the well. The drive-in is implemented by calling the *oxide-etch*, *strip-resist*, and *dry-oxide* procedures. Notice that the wet and dry oxidations are performed on the wafers in the `product` and `nwell` lots.

The last operation in the outer step measures the oxide thickness. Most BPFL procedures shown in this example are taken from libraries of procedures that are used by many processes.

The information specified in the process-flow thus far will be of interest to most interpreters. However, some interpreters need information that is meaningless to other interpreters. BPFL provides a model to specify information in different views of the process-flow. For example, the *fabrication* view describes the information needed to manufacture IC's. Information needed to simulate the process that is not meaningful to the fabrication process is specified in the *simulation* view. For example, the SIMPL implant

```
step WELL-FORMATION: begin
  /* Implant well and drive it in. */
  with-lot ('product, 'well) do
    wet-oxide(time: {11 min}, temp: {1000 degC},
              thickness: {100 nm}, measure: 'well);
  end;
  pattern('NWELL);
  with-lot ('product, 'well) do
    implantation(dopant: #m(phosphorus), dose: {4e12 /cm^2},
                 energy: {150 keV}, simpl-depth: {300 nm});
  end;
  step WELL-DRIVE-IN: begin
    with-lot ('product, 'well) do
      oxide-etch();
    end;
    strip-resist();
    with-lot ('product, 'well) do
      dry-oxide(thickness: {300 nm}, temperature: {1150 C},
                time: {4 hr}, anneal-time: {5 hr});
    end;
  end;
  measure-oxide-thickness(
    pick-wafer('product), tag: "well", location: "well");
end;
```

Figure 4: Well formation step.

model must be given the junction depth.

A hierarchy of views is defined as shown in figure 5. An interpreter processes only the commands in the view in which it is interested or any parents of that view in the view hierarchy. For example, an interpreter that executes a process-flow in the *scheduling* view will see procedure calls in the *scheduling*, *fabrication*, and *bpfl* views.

A BPFL program specifies the view in which each operation is defined. An interpreter begins execution in a particular view. For example, the WIP interpreter begins execution in the *fabrication* view and a simulation interpreter begins execution in the *simulation* view. A control-structure primitive *viewcase* is provided to change the view. For example, the following *viewcase* specifies different operations for the *simulation* and *fabrication* views.

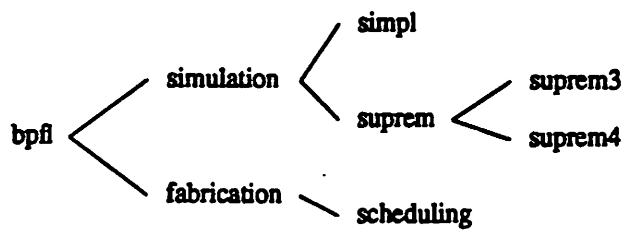


Figure 5: View hierarchy.

```
viewcase
  case simulation: simulation view operations;
  case fabrication: fabrication view operations;
end;
```

The code in the *simulation* view specifies the simulation operations that model the operations specified in the fabrication view. The *view* statement is provided to specify code for just one view as illustrated by:

```
view fabrication: fabrication view operations;
```

which specifies operations in the *fabrication* view.

Simulation input-generator interpreters must maintain a model of the current state of wafers as they are processed. Wafer state information is stored in a data structure that is similar to the external profile interchange format (PIF) used by most simulators. The definition of this data structure and examples of its use are given in section 5.

The code in the *fabrication* view specifies the operations required to make the IC. These operations allocate equipment and issue commands either directly to the equipment or to an operator who performs the operation. Examples of these operations are given in the next section.

4. Fabrication View

This section describes the BPFL abstractions provided to support IC fabrication. Figure 6 shows the software architecture of the WIP system. The WIP interpreter process (WIPIP) accesses the CIM database, communicates with operators through a user interface process (UIP), or communicates with equipment through an equipment interface process (EIP). The WIP interpreter and equipment interface processes are server processes. In other words, they handle many runs and equipment at the same time. In the Berkeley implementation, WIPIP is a Common Lisp program, EIP is the object-oriented SECS server developed by Wood [33], and UIP is an Application-By-Forms

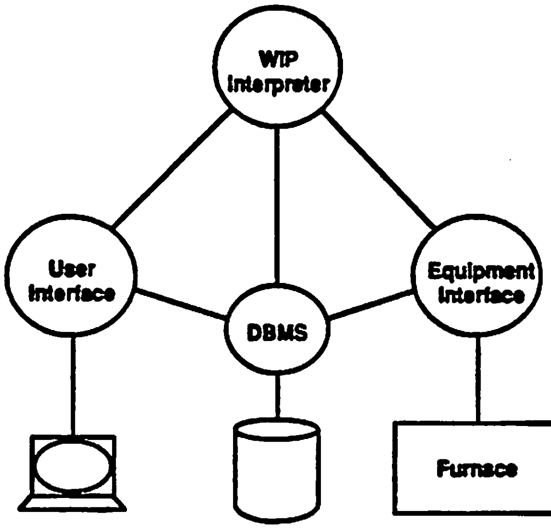


Figure 6: WIP system architecture

(ABF) program.[7] All of these processes access shared data stored in the CIM database.

The remainder of this section describes the abstractions to allocate and communicate with equipment, to communicate with operators, to log events and measurements, to specify constraints and rework, and to specify feed-forward and feed-backward control.

4.1. Equipment Abstractions

The equipment in a fab is described in the database so that equipment specific commands and settings can be separated from the commands to move lots, communicate with an operator, and perform other housekeeping operations. An object-oriented design is used to facilitate adding new types of equipment to the system. Figure 7 shows the equipment class hierarchy. In addition, the following class is defined that contains status information on all equipment

[7] ABF is a product of Ingres Corporation, Alameda, California.

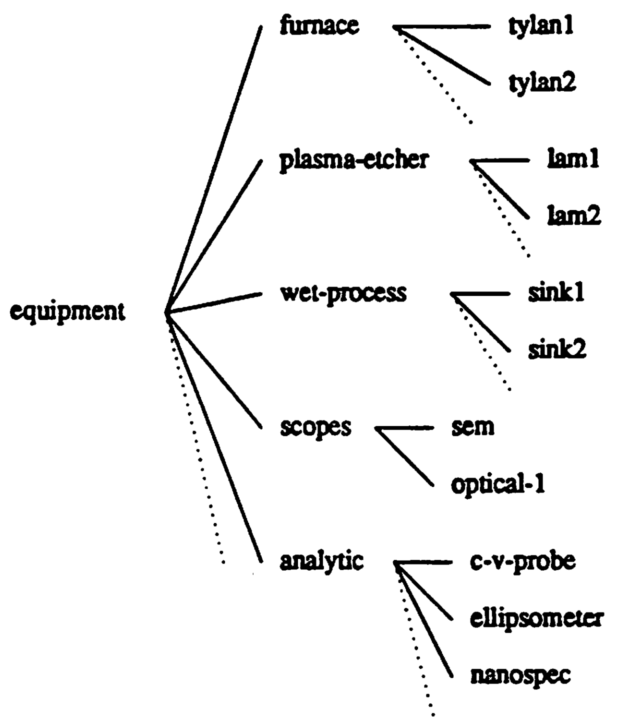


Figure 7: Equipment class hierarchy.

```
Equipment-Status(
    equipment: Equipment*
    status: (FREE, PROCESSING, OFF-LINE, WAITING-REPAIR,
            BEING-REPAIRED, BEING-MAINTAINED),
    allocated-to: datetime,
    estimated-time-available: datetime,
    time-last-used: datetime)
```

The *equipment* attribute contains a reference to an instance of an equipment object. The asterisk denotes that the reference can be to an equipment object or an object of a class that inherits from equipment (e.g., *tylan*).

Equipment is allocated to a run by the *with-equipment* construct, which takes the name of a specific piece of equipment (e.g., *lam-etcher-01*), the name of a category of equipment (e.g., *oven*), or a list of equipment names or categories and allocates a currently free piece of equipment that satisfies the specification. The equipment allocation function also implements the equipment reservation or scheduling policy [14,19,27]. A CLOS object that represents the allocated equipment is bound to a variable which can be used in the body of *with-equipment*. Other users are prohibited from using the equipment while it is allocated.[8]

After the particular piece of equipment has been allocated, operations are performed either directly by equipment operations or indirectly by communicating with an operator. Figure 8 shows a BPFL procedure that implements a hard bake operation that illustrates how equipment is allocated and direct execution operations are specified. The *hard-bake-resist* procedure takes one argument that selects either a standard or double-photo hard bake.

The fabrication view in the procedure body allocates an oven, downloads a recipe to the oven, executes the recipe, and deallocates the oven. The recipe in this example is specified with methods on the piece of equipment (*set-temp* and *wait*). These methods are compiled to SECS-II commands. If the equipment allows recipes to be downloaded dynamically, the code in the body of the *with-equipment* construct can use the database to track the recipes currently in the equipment and download the recipe when needed.

```
defflow hard-bake-resist(double-photo)
  "Hard bake resist on wafers."
begin
  viewcase
    case simulation: ...;
    case fabrication: begin
      with-equipment an-oven: 'oven do
        if double-photo then
            set-temp(an-oven, {150 degC});
            wait(an-oven, {30 min});
        else
            set-temp(an-oven, {120 degC});
            wait(an-oven, {20 min});
        fi;
      end;
    end;
  end;
end;
```

Figure 8: Equipment operation example.

[8] Functions are also provided so that fab managers can deallocate equipment through a facility management program should the need arise.

4.2. Operator Communication

The *user-dialog* function can be used to communicate with an operator. This function calls an ABF frame in the UIP. A frame contains a form through which data can be displayed to or edited by the operator and operations that he or she can execute [18]. The form contains data display fields that describe the operation to be performed and data entry fields in which the operator can enter measurements and status data. The frame includes operations the operator might need to execute such as abort an operation, operation completed, check equipment status, and disconnect from a run.

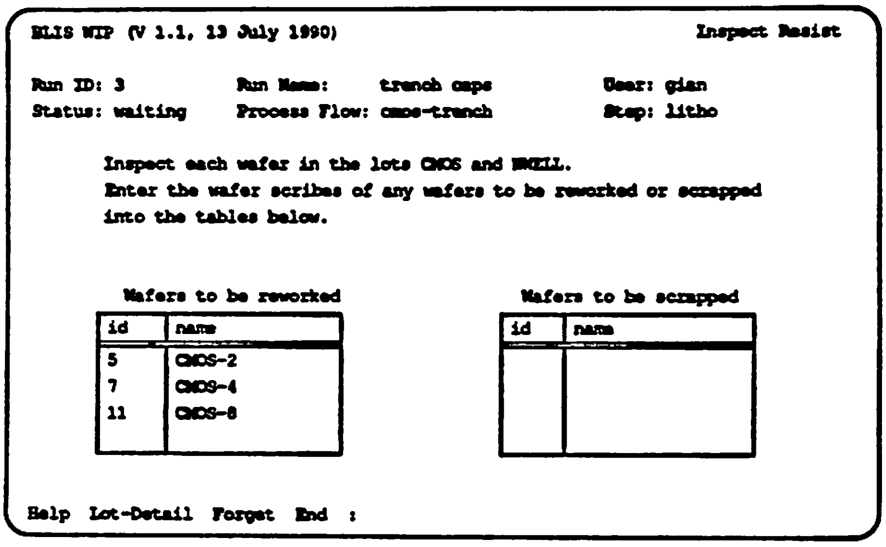
Figure 9 shows a procedure that communicates with an operator to inspect the resist on a wafer. The body of the procedure allocates a microscope and asks the operator to inspect each wafer in the lot. The operator enters wafer identifiers and inspection results into the form in the *inspect-resist* frame shown in figure 10 that is called by the *user-dialog* command. The data entered by the operator is returned to the procedure as a list of two lots: those to be reworked and those to be scrapped. The procedure moves the wafers from the `current` lot to either the `rework` or `scrap` lots. After examining the wafers, the microscope is deallocated.

4.3. WIP Log

A BPFL procedure can append records to the CIM database to log events that occur or measurements that are taken during processing. The log is represented by a sequence of

```
defflow inspect-resist()
  "Inspect each wafer and put wafers to be reworked into rework
   and wafers to be scrapped into scrap."
begin
  view fabrication:
    with-equipment scope: 'microscope do
      let results := user-dialog(name: 'inspect-resist,
                         tag: "inspect-wafer-resist", equipment: scope);
      begin
        move-sublot(first(results), 'current, 'rework);
        move-sublot(second(results), 'current, 'scrap);
        log('Resist-Inspection-LO, wafer-status: results);
      end;
    end;
end;
```

Figure 9: *User-dialog* function example.



```
BLIS WIP (V 1.1, 13 July 1990)                                  Inspect Resist

Run ID: 3           Run Name:       trench caps           User: gian
Status: waiting     Process Flow: cmos-trench             Step: litho

        Inspect each wafer in the lots CMOS and NWELL.
        Enter the wafer scribes of any wafers to be reworked or scrapped
        into the tables below.

         Wafers to be reworked                     Wafers to be scrapped
```

id	name
5	CMOS-2
7	CMOS-4
11	CMOS-8

id	name

```
Help  Lot-Detail  Forget  End  :
```

Figure 10: *Inspect-resist* frame.

CLOS objects of different classes. Each class represents a different type of event that is being logged. Figure 11 shows the class hierarchy. Figure 12 shows the database table

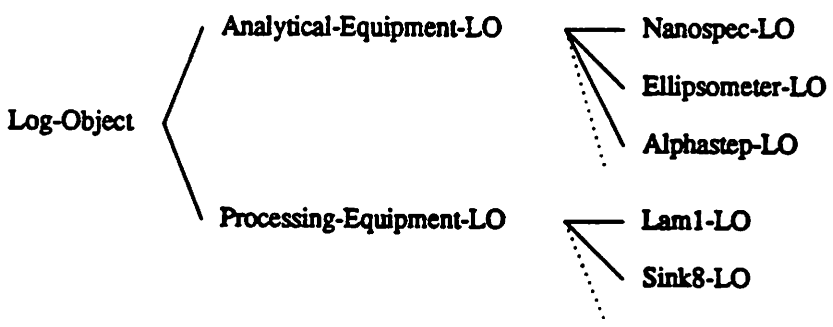


Figure 11: WIP log object class hierarchy.

definitions for the different types of log objects (LO).

The log is stored in the CIM database as a separate relation that contains information about each entry (e.g., the time the object was written and the run that wrote it) and a reference to the specific LO. The log object written in the *inspect-resist* procedure above was a *Resist-Inspection-LO* object which recorded the status entered by the operator for each wafer inspected.

The *WIP-Log* entry also includes attributes that allow a user to determine which operation in a process-flow wrote the entry. The *process-name* attribute specifies the process, the *procedure-name* attribute specifies the procedure, and the *step-path* attribute specifies the step that wrote the entry. The value stored in the *step-path* attribute contains the current step names concatenated together to form a string. A path name is required because steps can be nested. For example, the *WELL-FORMATION* step shown in figure 4 contained a nested step *WELL-DRIVE-IN*. Consequently, the step path for a log entry written in the *WELL-DRIVE-IN* step is "WELL-FORMATION/WELL-DRIVE-IN."

The *time* attribute specifies the date and time when the object was written and the *tag* attribute records a value included in the command that writes the log object. The *tag* attribute can be used to specify a unique string that identifies the log objects written by a particular command. This string can be used to simplify the predicate required to search the log for all objects written by the command.

WIP Log Class
```
WIP-Log(run-id: integer, log-object: Log-Object*, process-name: string,
        procedure-name: string, step-path: string, time: datetime,
        tag: string)
```
Analytic Equipment Log Objects
```
Log-Object()
Nanospec-LO(t: unit[]) inherits (Log-Object)
Ellipsometer-LO(t: unit[]) inherits (Log-Object)
Alphastep-LO(height-array: unit[]) inherits (Log-Object)
Resist-Inspection-LO(wafer-status: string) inherits (Log-Object)
  ...
```
Processing Equipment Log Objects
```
Lam1-LO(wafer-etch-time: datetime) inherits (Log-Object)
Sink8-LO(etch-time: datetime, resistivity: unit) inherits (Log-Object)
  ...
```

Figure 12: Database schema for WIP log objects.

The *log* function is provided to write log entries. The arguments to *log* include the log object class to be written and a collection of class specific arguments that record the desired data. For example, the log command in the *inspect-resist* procedure above is:

```
log('Resist-Inspection-Log, wafer-status: wafer-inspections)
```

The first argument is the log object class and the class-specific argument is an array of wafer status data.

The log can be queried to fetch arbitrary sets of log objects that can be analyzed to determine what happened when the process was run. Queries can be executed from an ad hoc query interface, an engineer's notebook interface [1], or a BPFL program. The engineer's notebook interfaces allows a user to browse the log and create hypertext links to particular entries. A program can access the log to make decisions based on it's past history without having to create special data strutures to save the desired data. In other words, the log acts as an extensible data structure for recording information about the run that can be queried by the process-flow itself.

Figure 13 shows three sample log queries specified in an extended version of SQL that can be run. The first query retrieves all log objects written for a specific run. This query creates a data set about the run that can be further analyzed. The second query shows

```
/* retrieve log entries for run 132 */
    select *
    from WIP-Log
    where run-id = 132

/* retrieve resist-inspect measurements for CMOS-16 runs in
    the past 30 days */
    select log-object.wafer-status
    from WIP-Log
    where process-name = "CMOS-16" and time ≥ today() - "30 days"
      and class(log-object) = "Resist-Inspect-LO"

/* calculate the average number of ellipsometer entries written for
    each run since the beginning of the year */
    select average(log-object)
    from WIP-Log
    where time ≥ "1 January 1990"
      and class(log-object) = "Ellipsometer-LO"
    group by run-id
```

Figure 13: Sample log queries.

how particular log entries for a collection of runs can be retrieved. The last query shows how the log can be queried to determine statistics about equipment usage.

4.4. Constraints and Rework

Process-flows often require the specification of time constraints between operations (e.g., the time between exposing and developing the resist in a lithography operation must be less than one hour) and rework loops. BPFL provides functions to implement both features. These functions are typically of interest only to the WIP interpreter and scheduler because they deal with fabrication issues.

Constraints are specified as a scope over which certain predicates must be true. If a constraint is violated, the interpreter raises an exception that must be caught by a handler that knows how to deal with the situation. The proposed ANSI standard Common Lisp conditions package is used to define exception handlers and raise exceptions.

The *constrain* statement specifies the constraint, the action to take if the constraint is violated, and a sequence of statements over which the constraint must hold. For example, the following constraint suspends processing if the temperature in the fab goes above 72 degrees farenheit.

```
constrain
case current-temperature > {72 degF} then suspend-run();
begin
  statement;
  statement;
  ...
end;
```

The case clause specifies a constraint predicate which is followed by a **then** clause that specifies what to do if the constraint is violated. Several **case-then** clauses can be specified in one **constrain** statement. The WIP interpreter implements environmental constraints by passing them to an environment monitor that will notify the WIP interpreter if the condition is violated. After posting the constraint, the WIP interpreter continues execution of the operations in the *constrain* body. The constraints are removed when the execution of the body is completed.

A time constraint is more complicated. The following code specifies constraints on the time between lithography operations:

```
contrain
case ((max-time-between('spin-on-resist, 'expose-resist) > {2 days})
  or (max-time-between('expose-resist, 'develop-resist) > {1 hour}))
  then signal('time-constraint-violation);
begin
  spin-on-resist();
  expose-resist(mask-name: mask);
  develop-resist();
  hard-bake-resist(double-photo: true);
end;
```

The body of this statement spins, exposes, develops, and hard-bakes the resist. Time constraints are defined between spinning and exposing the resist (two days) and between exposing and developing the resist (one hour). The arguments to the *max-time-between* function are function names. *Max-time-between* returns the time between the last equipment operation executed in the first function and the first equipment operation in the second function. This function is only useful in *constrain* predicates because it is not a true function.

Time constraints are implemented by the WIP interpreter. It maintains a data structure that keeps track of equipment operations called from functions within a constraint body.[9]

Constraints are frequently used inside a rework loop. The rework loop specifies the processing to be done, a test for correctness (e.g., inspecting the resist with a microscope), and operations to execute if one or more wafers fail the test. The *rework* function takes the following arguments:

(1) the operations to perform (*body*),

(2) a procedure to test whether the operations were successful (*test*),

(3) the number of times to retry the loop before giving up (*retries*),

(4) operations to perform before retrying the operations in the body (*rework-prefix*), and

(5) a function to call if the retry count is exceeded (*retry-failure*).

The semantics of the *rework* function are shown in figure 14. The *test* function tests the wafers and puts the ones that can be reworked into `rework` and the ones that cannot be reworked into `scrap`. The loop is exited if the `rework` lot is empty. Otherwise, the rework prefix is executed and the rework wafers are reprocessed.

Figure 15 shows the definition of the *pattern* procedure which uses a rework loop, a constraint, and the *inspect-resist* procedure described above. Notice that the *contrain* discussed above is used in the body of the rework loop to perform the patterning operation. The only difference is that instead of raising a general exception the *force-rework* exception is raised which is caught by the *rework* statement. This exception causes all wafers to be reprocessed as though they failed the test after processing.

4.5. Control and Data Structures

BPFL is embedded in CL so the full complement of Lisp control and data structures can be used. For example, a BPFL procedure can store a measurement in a variable which can be tested later in the run to change the processing on the lot. In other words, simple programming constructs are used to specify feed-forward control. Feed-backward

[9] Functions are provided so that equipment operations performed by an operator using a *user-dialog* function can also be logged correctly.

```
let retry-count := retries;
begin
  with-lot ('current) do
    while true do
      body;
      test;
      deallocate-wafers('scrap):
      if null(lot-indexes('rework)) then return; fi;
      if zerop(retry-count) then retry-failure; fi;
      retry-count := retry-count - 1;
      set-current-lot('rework);
      rework-prefix;
    end;
  end;
end;
```

Figure 14: Semantics of *rework*.

```
defflow pattern(mask, will-double)
  /* Pattern a positive resist. */
begin
  viewcase
    case suprem: ...;
    case fabrication: begin
      rework
        contrain
        case ((max-time-between('spin-on-resist, 'expose-resist)
                        > {2 days})
          or (max-time-between('expose-resist, 'develop-resist)
                        > {1 hour}))
          then force-rework();
        begin
          spin-on-resist();
          expose-resist(mask-name: mask);
          develop-resist();
          hard-bake-resist(will-double: will-double);
        end;
        test inspect-resist();
        retries 5;
        rework-prefix strip-resist();
      end;
    end;
  end;
end;
```

Figure 15: *Pattern* procedure definition.

control is specified by having the procedure query the database for information stored there by previous runs.

This section described the functions provided to execute equipment operations, to communicate with the operator, to write log entries, and to control the flow of execution. The next section describes the BPFL functions provided to specify wafer state changes.

5. Simulation View

This section describes the BPFL abstractions used to specify information required by simulation input-generator interpreters and an example of generating SIMPL input. BPFL provides abstractions for material specifications, masks, and wafer profiles.

Figure 16 shows the class hierarchy for materials. These classes have attributes that describe properties of the material and names that simulators use for the material. For example, the class *poly* has the following attributes: 1) *crystal* that specifies that poly has

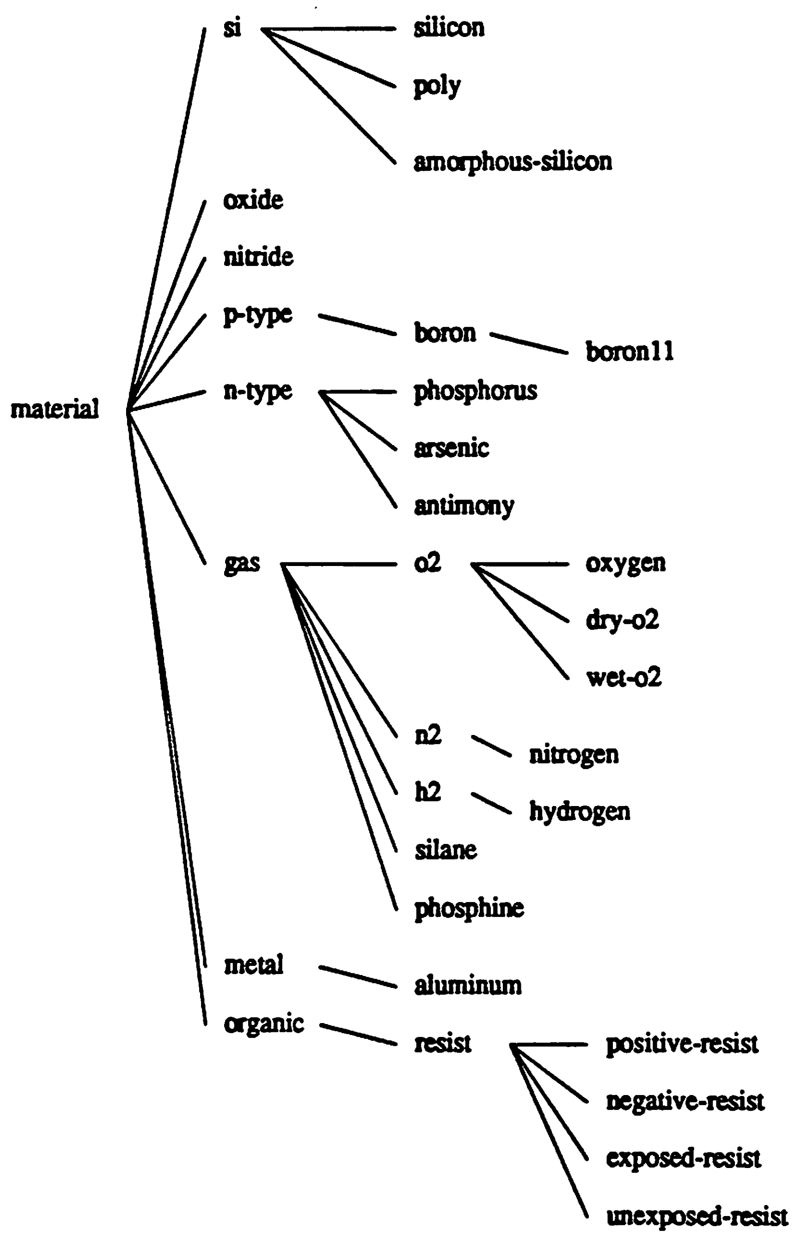


Figure 16: Material class hierarchy.

a crystal structure, 2) *grain-size* that specifies the grain size of the poly material, and 3) *simpl-name* that specifies the name for poly used by the SIMPL simulator. Material objects are used to specify materials used in an operation.

Masks are represented by a class that includes two attributes: *name* and *location*. *Name* specifies the external name of the mask (e.g., NWELL). *Location* specifies a set of design layers that are used to make the mask. *Location* values are expressed as predicates that indicate which regions of the mask are clear and dark. For example, the value *NOT layer-1* specifies that the colored area(s) in the design layer *layer-1* are inverted, which means those areas are clear in the mask. The boolean operator *OR* specifies area union and the operator *AND* specifies area intersection. Mask objects are used as arguments to procedures that perform lithography operations such as *pattern* in Figure 15.

A profile interchange format (PIF) abstraction is provided by BPFL to represent the stacking of materials in a wafer profile. A data structure is maintained that contains a collection of *snapshots* that represent wafer profiles. The structure is optimized so that only one copy of the information is maintained for wafers with the same profile. Operations are provided to change snapshots to simulate the effects of processing operations (e.g., depositing material, etching material, and removing material).

Figure 17 shows a block diagram of a wafer profile that contains three layers: silicon, oxide, and resist. The area above the resist layer is the ambient air. Figure 18 shows a graphical depiction of a PIF snapshot for this block diagram. Snapshots are composed of segments, boundaries, and attributes. Segments specify the information about a region or layer in a profile. They are represented by labeled ovals in the graphical depiction. The snapshot has three segments that correspond to the three layers in the block diagram. The ambient segment is a built-in segment supplied by the system for each snapshot.

A boundary specifies that one segment is on top of or next to another segment. Boundaries are represented by solid lines in the graphical depiction. The profile above has

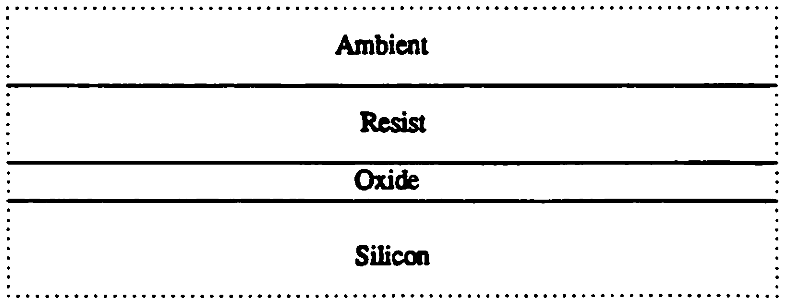


Figure 17: Block diagram of a wafer profile.

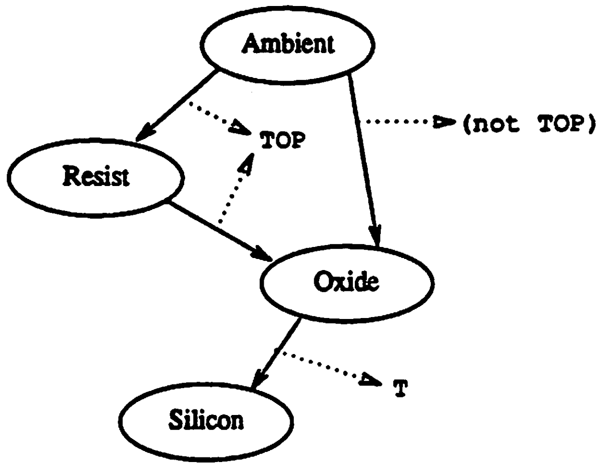


Figure 18: Graphical depiction of the PIF structure for the block diagram in figure 17.

three boundary lines that represent that each layer is on top of another. The dotted arrows point to location predicates that specify where the boundaries exist. For example, boundary between oxide and silicon everywhere so the predicate is *T* which represents **true** in Lisp. Oxide is exposed to the ambient on the top and backside of the wafer as specified by the two boundary lines that exit the *Ambient* oval.[10]

Attributes are used to specify properties about the profile (e.g., material name, whether a resist is exposed, boundary locations, etc.). They are represented by pairs of keywords and values. Attributes can be attached to segments, boundaries, and other attributes.

Functions to manipulate PIF snapshots are provided that correspond to typical operations in a process-flow. Figure 19 is a partial list of these functions with a short description of their semantics. For example, the function

```
        etch-material-in-lot(#m(resist, negative: nil, exposed: true),
                                          true)
```

changes the PIF structure of all wafer snapshots in the lot to etch the exposed resist. These functions are used to model changes in wafer profiles as a lot is processed.

Figure 20 shows the *expose-resist* procedure called from the *pattern* procedure defined in figure 15. This procedure takes a mask name argument which specifies the mask to use

[10] The block diagram in figure 17 only depicts the material on the top of the wafer.

Function	Description
find-segments	Return a list of segments that have the desired properties.
find-surface-segments	Restrict find-segments to exposed segments.
set-pif-attr-in-lot	The given attribute is added to the given segments in all snapshots.
deposit-in-lot	A segment with the given attributes is added to all snapshots. Its location may be restricted to the top side.
split-segments-in-lot	Segments that overlap the given location are split in each snapshot. One new segment is within the given location, the other is outside.
etch-material-in-lot	All surface segments made of the given material are etched down completely where they are exposed.

Figure 19: Higher level functions to manipulate PIF structures.

```
defflow expose-resist(mask-name)
let mask := find-mask(mask-name);
      location := mask-location(mask);
      exposure-location :=
        intersect-layers(top-side(), invert-layer(location));
      newsegments;
begin
  viewcase
    case simpl: ...;
    case fabrication:
      with-equipment e: 'exposure-machine do
        viewcase
          case sample: ...;
          case fabrication:
            user-dialog('align-and-expose, mask: mask);
        end;
      end;
  end;

  /* modify PIF snapshot */
  new-segments := split-segment-in-lot(
                     find-segments(material: #m(unexposed-resist)),
                     exposure-location);
  modify-segment-material-in-lot(new-segments, exposed: true);
end;
```

Figure 20: *Expose-resist* procedure definition.

in the patterning operation. The procedure has four local variables: 1) the mask object (*mask*), 2) a specification of the dark areas on the mask (*location*), 3) a specification of the clear areas on the mask (*exposure-location*), and 4) a variable to hold the list of new segments (*new-segments*); The clear area is calculated from the dark area specification given in the mask object. The body of the procedure executes the expose operation and modifies the PIF snapshot. The commands to change the PIF snapshot follow the fabrication command so that the changes will not have to be undone if the equipment operation fails. Notice that simulation view commands are mixed with fabrication view commands to specify parameters that are specific to a particular simulator (e.g., SAMPLE and SIMPL).

The commands that change the PIF snapshot split the resist segments into exposed and unexposed segments. Exposed segments are ones on the top of the wafer where the mask is clear. Figure 21 shows a block diagram of the wafer profile after the *expose-resist* procedure has been executed on a wafer with the profile shown in figure 17.

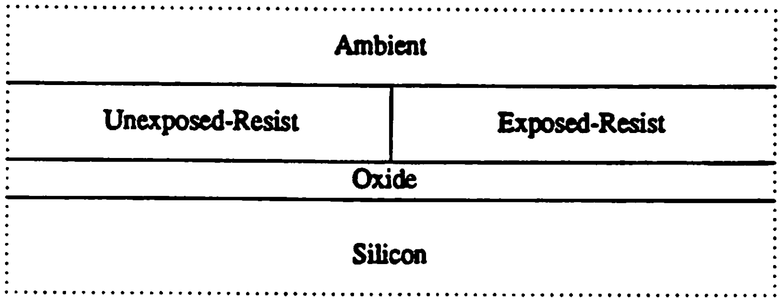


Figure 21: Block diagram after *expose-resist* procedure.

Figure 22 shows the definition of the *develop-resist* procedure called in the *pattern* procedure. This procedure contains commands that specify operations in three views: *simpl*, *sample*, and *fabrication*. The *simpl* view operation specifies the SIMPL simulator command for a develop operation. The *fabrication* view operation executes a particular

```
defflow develop-resist()
begin
  viewcase
    case simpl: simpl-op("DEVL", "ERST");
    case sample: ...;
    case fabrication:
      with-equipment e: 'mti-developer do
        develop-recipe(e);
      end;
  end;
  etch-material-in-lot(
    #m(resist, negative: nil, exposed: true), true);
  etch-material-in-lot(
    #m(resist, negative: true, exposed: nil), true);
end;
```

Figure 22: *Develop-resist* procedure definition.

recipe on a developer. The two function calls at the end of the procedure remove the exposed positive resist and the unexposed negative resist from the PIF snapshots. Figure 23 shows a block diagram of the wafer profile after the *develop-resist* procedure has been executed on a wafer with the profile shown in figure 21. Notice that the exposed segments in the block diagram have been removed.

The SIMPL input-generator interpreter produces the following code when it is run on the *expose-resist* and *develop-resist* procedures given the wafer profile shown in figure 17:

> EXPO *mask-name* no ERST
> DEVL ERST

The *EXPO* command is generated when the *expose-resist* procedure is interpreted and the *DEVL* command is generated when the *develop-resist* procedure is interpreted. The *mask-name* in the *EXPO* command is replaced by the particular name of the mask passed to the procedure. The *no* argument specifies that the mask should not be inverted. This value is derived from the mask object. The *ERST* argument specifies the SIMPL name for exposed resist. The *DEVL* command takes one argument, which specifies the resist that should be developed.

This section described the BPFL abstractions supplied to specify information required by simulators.

6. Discussion

This section comments on the size and complexity of process-flow specifications, describes the status of the BPFL interpreters that have been implemented, and discusses the infrastructure required to implement a CIM system like the one described here.

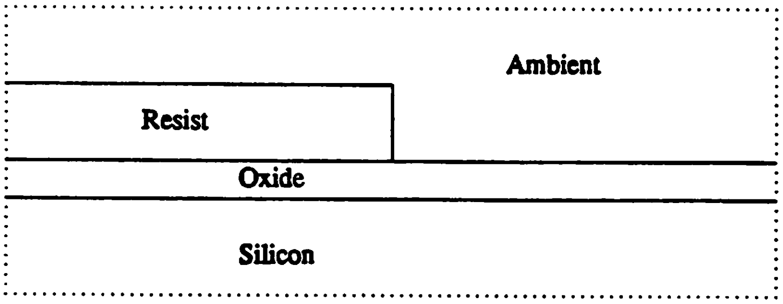


Figure 23: Block diagram after *develop-resist* procedure.

A BPFL specification for the Berkeley CMOS-16 process has been completed. The specification is approximately 650 lines of BPFL code not counting the definitions of the 75 ABF frames used for operator interactions. The textual definition of this process is approximately 500 lines not counting the equipment recipes. While the BPFL specification is larger and more complex than the textual specification, remember that it is more complete and it can be used both for fabrication and simulation.

The size problem can be reduced by using libraries of predefined procedures. In a commercial fab that runs many processes, procedures can be developed that can be shared by the different process-flows. An important side-effect of using a formal process-flow language is that the design, implementation, and control of procedure libraries can be managed more easily. More importantly, process engineers can re-use procedures in the library and thereby shorten process development time. The full benefits of using a formal process-flow language will not be realized unless it is used with a good procedure library.

We have implemented a WIP and simulation input-generator interpreters. The WIP system is being tested now in the Berkeley Microfabrication Laboratory [28].

The simulation input-generator interpreter can generate the input needed by the PROSE simulator. We are in the process of completing the connection between the interpreter and the simulator.

BPFL programs can be decomposed into user-defined functions (e.g., *defflow* procedures and Lisp functions) and built-in functions. Built-in functions are either Lisp functions (e.g., *setf* or *let*) or functions supplied by an interpreter (e.g., *user-dialog* is supplied by the WIP interpreter). A Lisp code walker is used as the basis for the two interpreters. A particular interpreter is implemented by supplying definitions for the built-in functions defined for that interpreter.

The WIP interpreter just interpretes the BPFL program. The state of a run is checkpointed to the database when operations will take a long time (e.g., an equipment operation or an operator dialog). The run is retrieved from the database when the WIP interpreter receives input for it (e.g., the equipment or operator signals completion of an operation). Recall that the WIP interpreter is a server process so it switches between different runs. The WIP interpreter is approximately 10,000 lines of Lisp.

The simulation input-generator interpreter is slightly different. It walks the BPFL program and translates it into a Lisp program that will generate the simulation input when executed. This design simplified the implementation of the interpreter, which is approximately 9000 lines of Lisp.

Considerable infrastructure is required to implement a CIM system such as the one described here. Some parts of the system have been easy to implement (e.g., the CIM database and the WIP system user interface) because we could use commercial software packages. Other parts of the system were relatively easy to implement because we chose good tools (e.g., Lisp simplified the implementation of BPFL and the interpreters).

Some parts of the system have been very hard to implement. We have had considerable trouble implementing equipment connections and a graphical user interface to specify BPFL. Fortunately, tools are now available that should alleviate these problems. Wood's SECS server provides an excellent model for connecting equipment to the system and interfacing programs to it. And, the PICASSO graphical user interface development system will make it possible to experiment with forms- and icon-based interfaces for BPFL [20].

Lastly, the recent development of distributed programming tools for Unix[†] (e.g., OSF's Decorum and Unix International's Open Network Computing) should simplify the development of a CIM system. Good application program interfaces (e.g., remote procedure call) and other higher level services (e.g., program abstractions to register for a stream of monitoring data similar to the Dow Jones news wire feeds supported in Teknekron's software bus [21]) will further simplify CIM system development.

7. Conclusions

This paper described the design of a process-flow language that can be used to represent IC manufacturing processes. The novel features of the language are:

(1) integrated process specifications that can be used by different applications (e.g., WIP, simulators, etc.),

(2) a PIF model that is used to support simulation process correctness interpreters, and

(3) support for the full power of a programming language (e.g., data and control structures and exception handling).

We believe BPFL achieves the design goals established for the language. Examples were presented that illustrated how one specification can be used by different interpreters. And, examples were presented that showed how conditional processes and feed-forward and feed-backward control can be specified.

Two interpreters for BPFL have been implemented: a WIP system and a simulation input-generator. While further experimentation is needed, our experience to date with the language and these interpreters has been positive.

Acknowledgements

We want to thank our collegues Dave Hodges and Costas Spanos and their students who have provided encouragement, guidance, and help during the course of this work. We also want to thank Chi-Yung Fu, Yosi Schacham, Dan Wolfson, and our industrial

[†] Unix is a trademark of A.T.&T.

sponsors who have made numerous suggestions that have improved this work.

References

1. B. Becker, D. Mudie and L. Rowe, "A Paper-Free Replacement for an Engineer's Laboratory Notebook", To be presented at 1990 SRC/DARPA CIM-IC Workshop, Berkeley, CA, Aug. 1990.
2. D. Boning, "Gestalt", *unpublished manuscript*, 1989.
3. C. J. Date, *An Introduction to Database Systems (Volume II)*, Reading, MA: Addison-Wesley, 1984.
4. B. Davies, *Personal communication.*, Texas Instruments, Dallas, TX, July 1990.
5. C. R. Glassey and S. Adiga, "Conceptual Design of a Software Object Library for Simulation of Semiconductor Manufacturing Systems", *Journal of Object Oriented Programming*, Nov. 1989.
6. R. Hartzell, *Personal communication.*, Texas Instruments, Dallas, TX, Jan. 1989.
7. C. P. Ho, J. D. Plummer, S. E. Hansen and R. W. Dutton, "VLSI Process Modeling — SUPREM III", *IEEE Trans. Electon Devices ED-30*, 11 (1983), 33-46.
8. W. C. Holton and et. al., *Japanese Technology Evaluation Program Panel Report on CIM and CAD for Semiconductor Industry in Japan*, Science International Corporation, McLean, VA, Dec. 1988.
9. K. Lee and A. R. Neureuther, "SIMPL — 2 (SImulated Profiles from the Layout — version 2)", *Symp. VLSI Tech. Dig. Tech. Papers*, Kobe, Japan, May 1985.
10. S. Leeke, B. Davies and K. Saraswat, "The Virtual Wafer Fab Modeling System", *Proc. Symp. Automated IC Manuf., Elecctochem. Soc. Meet.*, Honolulu, HI, 1987.
11. S. Leeke, "A Graphical, Extensible, Object-Oriented Modeling System for Semiconductor Technology Modeling, Analysis, and Simulation", Talk given at 1989 SRC/DARPA CIM-IC Workshop, Ann Arbor MI, Aug. 1989.
12. M. B. McIlrath and D. S. Boning, "Integrating Process Design and Manufacture", *Proc. 1989 SRC IFM-IC Workshop*, College Station, TX, Nov. 1989.
13. D. C. Mudie and N. H. Chang, "FAULTS: An Equipment maintenance and Repair System", *Proc. The 1990 IEEE/CHMT International Electronics Manufacturing Technology Symposium*, Washington DC, Oct. 1990.
14. A. Najmi and C. Lozinski, "Managing Factory Productivity using Object-Oriented Simulation for Setting Shift Production Targets in VLSI Manufacturing", *Proc. 1989 SME AUTOFACT Conf.*, Detroit, MI, Oct. 1989.

15. H. L. Ossher and B. K. Reid, "Fable: A Programming-Language Solution to IC Procss Automation Problems", *ACM-SIGPLAN Notices Notices 18*, 6 (June 1983).

16. J. Y. Pan, J. M. Tenenbaum and J. Glicksman, "A Framework for Knowledge-Based Computer-Integrated Manufacturing", *IEEE Trans. on Semiconductor Manufacturing 2*, 2 (May 1989).

17. D. T. Phillips, G. L. Curry and B. L. Deuermeyer, "CHIPS: A Coherent and Integrated Planning System for Semiconductor Manufacturing Systems Analysis", To be presented at 1990 SRC/DARPA CIM-IC Workshop, Berkeley, CA, Aug. 1990.

18. L. A. Rowe, "*Fill-in-the-Form* Programming", *Proc. 11th Int. Conf. on Very Large Data Bases*, Aug. 1985.

19. L. A. Rowe and C. B. Williams, "An Object-Oriented Database Design for Integrated Circuit Fabrication", *Proc. 1st Intl. Conf. on Data and Knowledge Systems for Eng. and Manuf.*, Hartford, CT, Nov. 1987.

20. L. A. Rowe and et. al., "The PICASSO Application Framework", Electronics Research Lab. Memo 90/18, U.C. Berkeley, May 1990.

21. D. Skeen, *Personal communication.*, Teknekron Software Inc., Palo Alto, CA, Sep. 1989.

22. G. L. Steele, *Common Lisp - The Language*, Digital Press, 1984.

23. M. R. Stonebraker and L. A. Rowe, "The Design of POSTGRES", *Proc. 1986 ACM-SIGMOD Conf. on Management of Data*, Washington, DC, June 1986.

24. M. Stonebraker and et. al., "Third-Generation Data Base System Manifesto", *Proc. IFIP Conf. on Object-Oriented Databses*, Windermere, England, June 1990.

25. M. R. Stonebraker, L. A. Rowe and M. Hirohama, "The Implementation of POSTGRES", *IEEE Trans. on Knowledge and Data Engineering 2*, 1 (Mar. 1990).

26. S. Tang and E. Wood, "An Object-Oriented Design Toolkit for CIM", To be presented at 1990 SRC/DARPA CIM-IC Workshop, Berkeley, CA, Aug. 1990.

27. D. Troxel, *Personal communication.*, MIT, Cambridge, MA, Nov. 1989.

28. K. Voros and P. K. Ko, "Evolution of the Microfabrication Facility at Berkeley", Electronics Research Lab. Memo 89/109, U.C. Berkeley, Sep. 1989.

29. J. S. Wenstrand, I. Hiroshi and R. W. Dutton, "A Manufacturing-Oriented Environment for Synthesis of Fabrication Processes", *Proc. 1989 ICCAD*, Nov. 1989.

30. C. B. Williams and L. A. Rowe, "The Berkeley Process-Flow Language: Reference Document", Electronics Research Lab. Memo 87/73, U.C. Berkeley, Oct. 1987.

31. D. Wolfson, *Personal communication.*, Siemens Corporate Research, Princeton, NJ, Nov. 1988.

32. A. S. Wong, ''An Integrated Graphical Environment for Operating IC Process Simulators'', Electronics Research Lab. Memo 89/67, U.C. Berkeley, May 1989.

33. E. J. Wood, H. Schenck and J. Wijaya, ''Networking and Object-Oriented Coding for SECS Communications'', *Proc. Automated IC Manufacturing Symposium*, Fall Electrochemical Society Meeting,, Oct. 1987.