PROCESS-FLOW SPECIFICATION AND DYNAMIC RUN MODIFICATION FOR SEMICONDUCTOR MANUFACTURING

by

Christopher James Hegarty

Memorandum No. UCB/ERL M91/40

15 April 1991

PROCESS-FLOW SPECIFICATION AND DYNAMIC RUN MODIFICATION FOR SEMICONDUCTOR MANUFACTURING

by

Christopher James Hegarty

Memorandum No. UCB/ERL M91/40

15 April 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

PROCESS-FLOW SPECIFICATION AND DYNAMIC RUN MODIFICATION FOR SEMICONDUCTOR MANUFACTURING

by

Christopher James Hegarty

Memorandum No. UCB/ERL M91/40

15 April 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

This dissertation describes applications of a semiconductor process representation to a factory control system for a computer-integrated manufacturing (CIM) system. The control system, which includes work in progress (WIP) and run management systems, uses a distributed heterogenous database to store all information. The database contains information about the fabrication facility, processes, work in progress, test data, inventory, orders, personnel, and products.

The process representation used by the WIP system is the Berkeley Process-Flow Language (BPFL). BPFL is designed to allow all information about a process to be merged into a common specification. This information includes the equipment, recipes, and parameters used to manufacture semiconductors, resource requirements needed for scheduling, and modelling parameters required for process simulation. Different programs, called interpreters, read a BPFL program and perform a task. For example, a process-check interpreter reads a BPFL program and checks that it does not violate processing rules. The WIP system is another example of an interpreter.

BPFL is an object-oriented language with abstractions defined specifically for semiconductor manufacturing such as lots, wafers, material, equipment, and wafer profiles. The language provides control structures designed for common processing activities such as lot splits and merges, equipment and operator communication, timing constraints, conditional control-flow, and rework loops. An exception handling mechanism is provided to allow processes to respond to unexpected conditions (e.g., equipment failure). The language is designed to separate facility-specific information from the process specification to make it easier to change equipment in a facility or move processes between facilities. BPFL and the WIP system make implementation of feedback and feedforward process control possible because data is stored in the shared database and processes coded in BPFL can access the database.

The WIP system supports equipment interfacing for automatic recipe execution and monitoring. The system is software fault-tolerant so that computer system failures will not cause loss of data. The run-management system allows active runs to be modified. For example, process flows may be edited while a run is active, and lots may be moved between runs. A software version control system maintains libraries of process-flow procedures to control process revisions.

[This page intentionally blank]

Contents

[This page intentionally blank]

List of Figures

[This page intentionally blank]

List of Tables

[This page intentionally blank]

Acknowledgments

This research has been 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.

I would like to express my appreciation to all of the members of the Berkeley CIM and CAM groups, past and present, who made valuable suggestions that contributed to the work described in this dissertation. Particular thanks are due to Kuang-Kuo Lin, Norman Chang, and David Mudie. Thanks also to Lauren Massa for her aid with INGRES installation, and to Jeff Sedayao of Intel for his help in understanding the capabilities of commercial CIM systems.

Christopher Williams, the creator of BPFL and the author of the core interpreter used to implement the WIP interpreter, deserves special mention. His programming skills allowed me to concentrate on developing the BPFL language constructs for fabrication and implementing the user interface for the WIP system. Christopher provided many valuable insights that contributed greatly to the work in this dissertation.

My thanks also to the members of my dissertation committee: Professors Ronald Gronsky, David Hodges, and Lawrence Rowe. Professor Hodges was my research advisor during the initial two years of this work. His enthusiastic belief in CIM and his knowledge of semiconductor manufacturing were a tremendous aid in deciding what features were important for the work described in this dissertation. Professor Rowe has been my research advisor for the final two years of this research and my dissertation chairman. His knowledge of software systems, language design, and commercial software were invaluable in the design and implementation of BPFL and the WIP system.

I'd also like to thank my parents, Peter and Janette Hegarty, for their support and encouragement to do well at whatever I chose and to continue my education.

Finally, my thanks to my wife, Celia. Her encouragement, confidence and sacrifice made this dissertation possible.

[This page intentionally blank]

Chapter 1
Introduction

The goal of *computer-integrated manufacturing* (CIM) is to use computer and information management technology to integrate and automatically execute manufacturing operations. Two key elements of a CIM system are a shared, integrated, distributed database and a process-flow representation suitable for all manufacturing phases. This thesis describes the implementation of a *work-in-progress* (WIP) system using a powerful process-flow representation for semiconductor *integrated-circuit* (IC) manufacturing. This chapter describes the problem domain for IC process-flow representation, process representation in commercially-available CIM systems, contemporary research into semiconductor process representation, and process representation in other industries.

1.1 Problem Domain

The objective of CIM is to increase productivity, information accessibility, and flexibility to meet changing market or manufacturing conditions. Other objectives include improving the accuracy of data and information collected during manufacture and increasing the transportability of manufacturing process specifications. Benefits include better predictability and reproducibility, higher yields, lower inventories, better equipment utilization, and greater throughput.

For the purposes of CIM, manufacturing operations include design, fabrication, and testing. Although CIM is applicable to any manufacturing process, IC manufacturing is the focus of this dissertation.

IC manufacturing is a complex process and it is becoming more complex due to increasing process sophistication, shorter design cycles, and shrinking product lifetimes. Table 1-1 shows the increase in IC fabrication complexity in the decade from 1980 to 1990. The volume of process data collected per lot (25 wafers) during manufacturing has risen from 100 records to 10 000 records in the evolution from large scale integration (LSI) to ultra-large scale integration (ULSI). With so many data records required, some form of automatic collection and monitoring is essential. Today CIM systems are necessary for efficient manufacture of complex semiconductor products.

A process flow specifies the steps needed to manufacture a product. It is an important component of any complex manufacturing process. A process-flow in the IC industry is called a *semi-*

conductor process representation (SPR). The goal of an SPR is that it be complete and facility-independent. It should also be machine interpretable to aid in process automation. Moreover, an SPR should be applicable to all stages of manufacturing (i.e., design, fabrication, and testing).

Semiconductor companies may view process-flow representations that are cryptic and hard to understand as an asset because of the security they provide. A major problem with IC-CIM systems used in industry is that different SPR's are used for design and fabrication. Frequently, more than one specification is used in each domain. This redundant specification is not surprising, given that the interests and goals of process designers and production staff differ greatly. For example, process engineers are primarily concerned with developing processes that meet device specifications (e.g., propagation delay), whereas production staff are primarily interested in high yield. The unfortunate consequence is that the different specifications are often inconsistent, which can significantly reduce productivity. For example, device simulators used by process engineers require exact numerical information about devices to be simulated. The information supplied to production staff is derived from the input to the simulators but often changes are made to accommodate processing limitations and correct errors in the original process specification, and these changes are frequently not made to the original simulation-based specification. Also, the translation of process specification from design to fabrication is normally done by hand, which is both error-prone and time consuming. A more serious problem arises because some semiconductor companies minimize the interaction between process designers and production staff. Process designers often consider the design complete when a process description and equipment recipes are delivered to the production facility. The production staff must re-learn much that the process designers already know before

	LSI	VLSI	ULSI
Products size in bits (DRAMS)	16K	256K–1M	4M
Throughput (wafers/month)	10 000	30 000	50 000
Total process steps per lot	100	230–400	550
Number of equipment types	40	100	120
Equipment count	70	300	400
Number of process conditions	200	800	1 500
Volume of process data per month (data base records) for stable production	4×10^4	6×10^6	2×10^7
Data base records/lot	100	5 000	10 000

Table 1-1: Trends in wafer fabrication - LSI, VLSI, ULSI [1].

the process can be used to manufacture an acceptable product. Since processes tend to evolve as equipment is updated and more is learned about the behavior of the process, there is often a need to transfer the process back into design for further analysis to contribute to new process design. Consequently the same difficulties are encountered in reverse because process designers are unable to use production process-flow representations. A single representation is required to solve this problem.

Another impetus that drives research into process-flow specification is the rapid increase in the number of process steps indicated in Table 1-1. A crucial advantage of a computer-interpretable process-flow specification is that version control systems can be used to coordinate changes in process flows and code libraries can be used to encourage the re-use of process-flow code.

An SPR includes the sequence of process steps (i.e., processing operations and data collection operations) required to fabricate a product. It also specifies information about the fabrication facility (e.g., equipment specifications). An SPR specification of a process flow is used to operate a fabrication facility (or *fab*), as input to a simulator, and as input to an equipment scheduler. Consequently information to support activities other than fabrication must be specified. For example, equipment schedulers need to know the time required to perform each process step.

An SPR must be able to express control flow in the process steps making up a process flow. For example, processing may be dynamically changed on the basis of data collected during prior processing of other lots on the same piece of equipment (i.e., feed-back control) or on the basis of data collected during prior processing of the same lot (i.e., feed-forward control). Systems that employ feed-forward and feed-back control are often referred to as *control-loops*. Control loops allow process designers to reduce manufacturing variability caused by equipment variations.

An important example of the use of control loops is photolithography in which there are significant interactions among physical parameters such as mask dimensions, photoresist exposure time, etching time and gate length in *Metal-Oxide-Semiconductor* (MOS) processes. A small deviation in an earlier step can be corrected by adjusting parameters in a subsequent step. Examples of control loops in photolithography are shown in Figure 1-1. In conventional photolithography *critical dimension* (CD), usually gate width in MOS, is measured. This measurement is used to adjust the etching step for subsequent lots. A CIM system that employs both feed-forward and feed-back

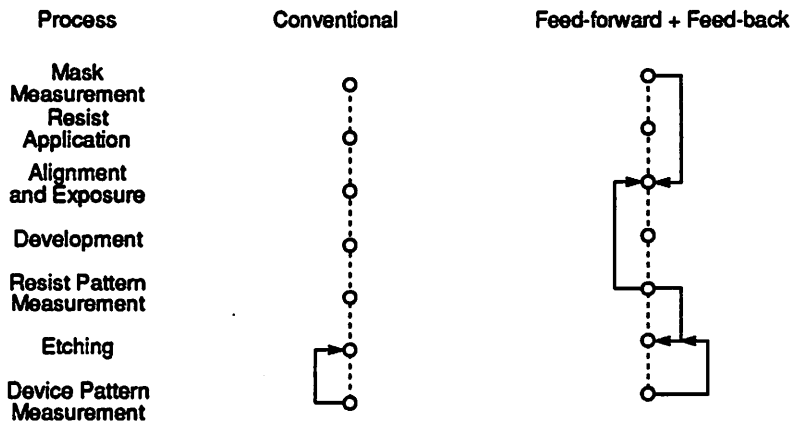

Figure 1-1: Control loops in lithography.

control can achieve more control than a conventional system. Feed-forward control can be used between the measurement of the mask CD and the exposure time, and again between the measurement of the resist pattern and the etching step. Similarly, feed-back control can be used to adjust the exposure time of subsequent lots based on resist pattern measurements and to adjust the etch time based on device pattern measurements. An example of the results obtainable with control loops is shown in Figure 1-2. Critical dimension control was improved by nearly 40 percent because small errors could be detected and corrected for during processing. In practical terms, an improvement of this magnitude is important because it reduces the percentage of parts with incorrect CD. Consequently, because CD is a primary factor in determining propagation delay for MOS devices and faster parts attract premium prices, significant improvements in profitability can be expected.

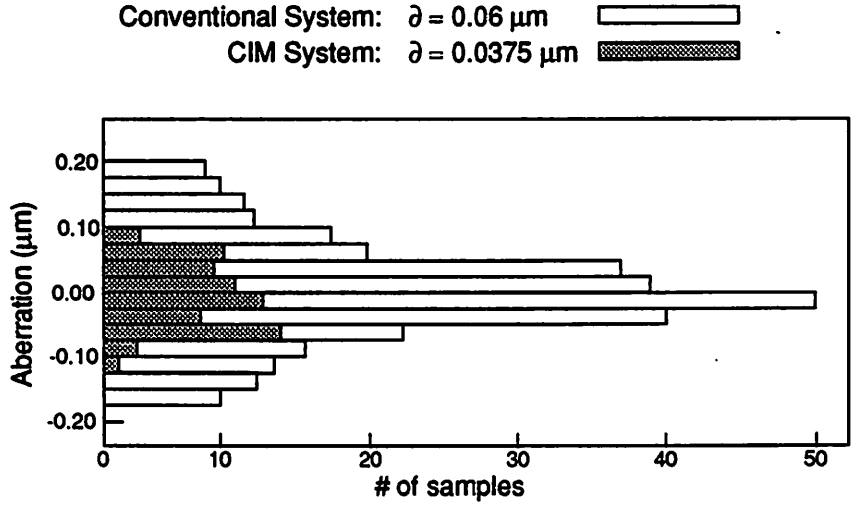

Figure 1-2: Critical dimension control in lithography [1].

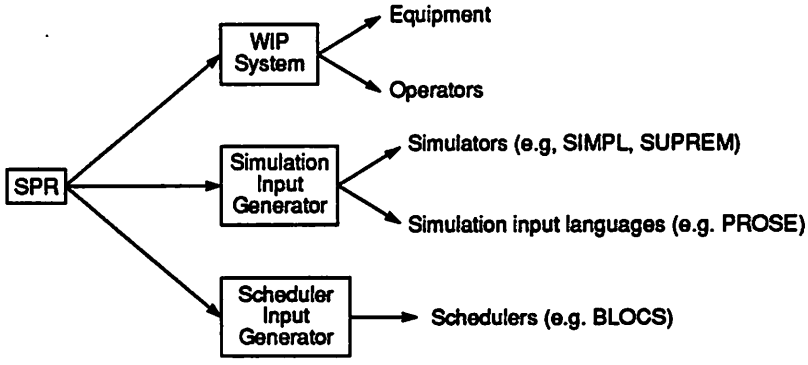

Figure 1-3: Information flow in SPR interpreters.

An extreme example of control-loops is the run-by-run control regime that is being explored by Sachs [2]. Feed-back control can only improve subsequent lots, based on the assumption that nothing has changed since the measured lot was processed. Feed-forward control, on the other hand, is particularly important for rapid ramp-up time (i.e., the time it takes a new process to achieve acceptable yields), which is essential for economic fabrication of small product quantities. The use of feed-forward control in a production facility requires real-time linkage of lot history and engineering data, since the information fed forward may be required almost immediately after it is collected. Support for feed-forward and feed-back control is an essential part of an SPR.

Although SPRs are intended to be used in many ways (e.g., shop floor control, simulation, scheduling), most prototype SPRs are developed for one application initially and extended to other uses later. One application of an SPR is in a work-in-process (WIP) system. A WIP system is responsible for handling operations concerned with the fabrication of products. It controls and records the movement of production lots through the fab, issues instructions to equipment operators to execute processing steps (or instructs equipment to execute steps automatically when possible), allocate resources (e.g., equipment, tracks the inventory of consumables), and maintains a log of the processing history of products.

An SPR interpreter *executes* a specification to accomplish a goal. Different interpreters accomplish different goals by performing different computations on the same specification as shown in Figure 1-3. For example, commands are issued to people and equipment when a WIP interpreter executes a process flow. Another interpreter will produce input commands for a process simulator

(e.g., SIMPL [3], SUPREM [4]) or a simulation input language (e.g. PROSE [5]) when it executes the same process flow. A scheduling interpreter generates run timing information for use by a scheduling system (e.g., BLOCS [6]).

1.2 The Berkeley CIM System

The development of the system described in this dissertation has been influenced by our vision of a CIM system architecture. This section describes that architecture. The system runs in a distributed heterogeneous computing environment composed of a variety of computers connected by a local-area network. A typical fab might use 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 1-4 shows a typical system. Notice that cell controllers have local databases and that 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. Equipment is connected to the cell controllers.

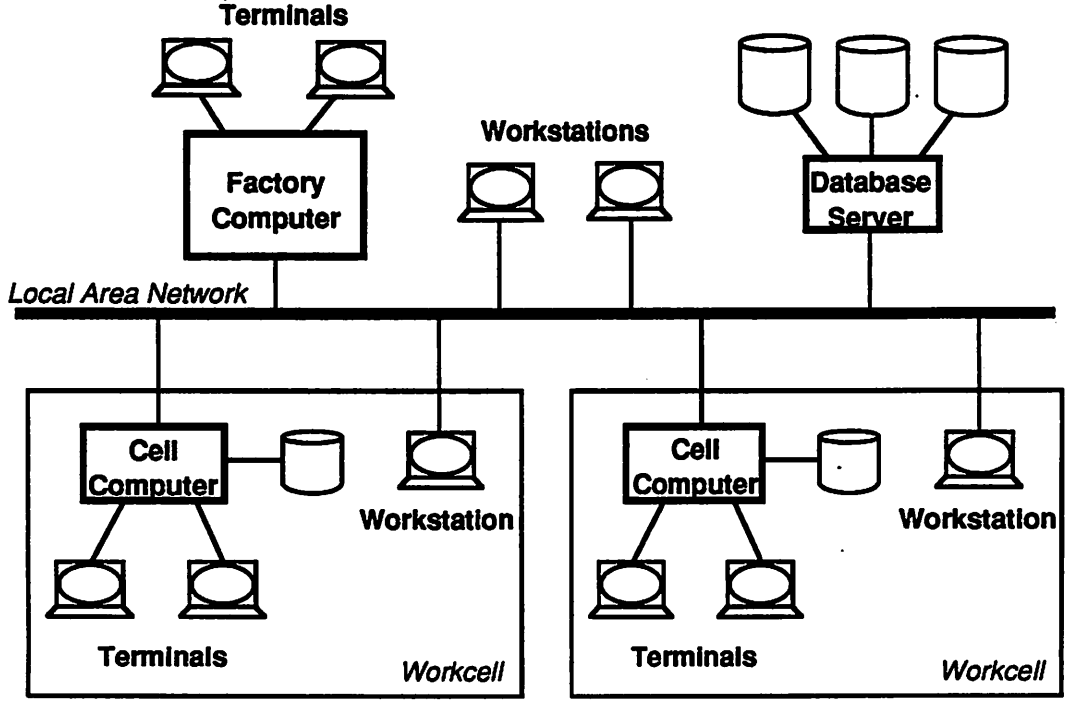

Figure 1-4: Typical IC-CIM fab computing system.

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 interprocess communications protocol.

A key component of the system is a shared CIM database that stores all the information about the design and manufacture of semiconductors. This database contains information about the manufacturing 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 [7], or INGRES/ STAR [8]). Data will be stored 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 heterogenous 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 into this architecture existing applications that use older technology storage systems (e.g., VSAM files and IMS databases). It would be prohibitively expensive to rewrite all applications because a new CIM system was introduced into the fab. However, new applications often must access the data managed by older applications. A distributed heterogeneous DBMS that provides gateways to different storage systems will allow fab applications to access new and old data. Consequently, data will be stored in many different systems including third generation database system, conventional database systems (e.g. relational, network and hierarchical), and files where appropriate.

A third generation database system is required for many CIM applications. It supports relational data storage and access, an object-oriented data model (i.e., inheritance, user-defined data

types, and methods), and a rules system [9]. An example is the POSTGRES system being developed at Berkeley [10]. A third generation database system 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 third generation database system can store arrays of user-defined data types (e.g., values with unit designations) in a table. Conventional DBMS's do not support these data types.

Although some attempts have been made to use existing programming languages as SPRs, these efforts have been largely unsuccessful. There are several problems with this approach:

1. *Robustness*. Since fabrication runs take a long time (e.g., a typical run lasts for more than a month), it is likely that a computer system failure will occur before a run is completed. Consequently, all state information about a process must be saved in non-volatile storage (e.g., on disk) so that a run can be restarted from its last saved position when the system crashes. Conventional programming language implementations are not designed with this form of recovery in mind.[1]
2. *Expressability*. Some common operations in semiconductor processing are difficult to express in a conventional programming language (e.g., timing constraints on steps).
3. *Dynamic Modification*. Fabrication processes run for weeks or months and it is frequently necessary to modify a process during a run. Such changes are difficult to make in most programming environments.

Although each of these shortcomings in conventional languages can be solved with sufficient effort, there is substantial benefit to be gained from using a special-purpose language developed specifically for SPR. This approach does not preclude translation of the SPR into a conventional language as a lower-level representation of the process flow.

[1] Some computer vendors have developed custom languages and programming styles to implement fault-tolerant programs in an On-Line Transaction Processing (OLTP) environment (e.g., Tandem [11]).

The SPR used in the Berkeley CIM system is the Berkeley Process-Flow Language (BPFL). BPFL is a procedural representation of a process flow. Process flows are represented as textual specifications of the sequence of steps required to manufacture a product.

The CIM database is used by BPFL and its associated interpreters in several ways. First, BPFL programs themselves are stored in the database. A software version-control system is implemented on top of the 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 [12]. A 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 computer failure. Mirrored disks, on-line backup and recovery, and standby spare databases can reduce the possibility that information is lost and reduce the down time should a failure occur [13].

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, in-process and in-situ measurements collected during processing, and other processing information is stored in the database. This log can be accessed by a BPFL procedure to change future processing based on previously recorded measurements (i.e., feed-forward or feed-back control). In other words, the database is a convenient repository for data that is shared within a run and between runs.

1.3 Commercial CIM Systems

Traditional SPRs in the semiconductor industry are based either on structured documentation or run-sheet specifications. They are typically used only for shop-floor control (i.e., WIP systems).

In a *structured documentation* system, a printed copy of the process-flow specification, called a *traveller*, accompanies the lot carrier and is passed along with it to different workcells.[1] Operators follow the instructions on the traveller that describe how the lot should be processed. As

[1] Wafers are processed in *lots* (25 wafers). Lots are transported around the fab in a *lot carrier*. Operations are performed at workcells, which are clusters of one or more pieces of fabrication equipment.

```
1.0 Starting Wafers: 18-22 ohm-cm, p-type, <100>
    Control wafers: NWELL (p-type), NCH (p-type)
    Measure bulk resistivity (ohms-cm) of NWELL on sonogage.
    R = _________

2.0 Initial Oxidation: target = 1000 A
      2.1 TCA clean furnace tube.
      2.2 Standard clean wafers:
          piranha 10 minutes, 10/1 HF dip, spin-dry.
      2.3 Wet oxidation at 1000 C:
             5 minutes dry O2
            10 minutes wet O2
             5 minutes dry O2
            20 minutes dry N2
          tox = __________
```

Figure 1-5: An example of structured documentation specification. This example shows the first two steps of the Berkeley `cmos-16` process flow [14].

each step is completed, the operator indicates that the step was completed and enters data for the step (e.g., measurements). Most structured documentation systems use computers to store specifications, but they normally do not possess the capability to interpret the specifications or collect data during a run. An example of structured documentation is shown in Figure 1-5. Note that spaces are available for an operator to write the results of measurements.

A *run-sheet* system is essentially an interactive traveller. The process-flow specification is stored in a computer and executed by it. When executed (or "interpreted"), the run-sheet describes the processing at each workcell, usually in the form of instructions displayed to the equipment operator on a computer terminal, and indicates where the lot should be moved when the step is finished. Some run-sheet systems issue commands to execute recipes stored in microcomputers that control the equipment and direct a material movement system to move lots to different workcells [1].

Examples of commercial run-sheet systems are WORKSTREAM [15] and PROMIS [16]. Both systems use a similar notation for process flows. In WORKSTREAM, the basic unit of a process flow is an *operation* which is defined as a single process step executed by an operator (e.g., "run the SWETOXB recipe in the furnace"). Sequences of operations are grouped into *routes*. A *product* is a sequence of one or more routes required to manufacture an item. The basic unit of a PROMIS process flow is a *recipe*. Recipes are subdivided into atomic operations which may be

shared among recipes (e.g., "set the temperature dial to 120 °C"). PROMIS has constructs called *processes* and *devices* which correspond to WORKSTREAM routes and products, respectively. While WORKSTREAM and PROMIS differ in the details of information specified at each level of abstraction, the representations are very similar in scope and power. The SPR provides operator input/output, material movement, and data collection and archiving commands. There is limited support for control flow, no exception-handling, and no support for activities other than fabrication.

Structured documentation and automated run sheet systems do not automate fabrication. They track WIP and provide production management information but they do not support equipment integration. Run-sheet specification languages are more powerful than structured documentation systems because they automate some operations, but they are typically limited to a small, fixed set of commands that are sequentially executed. These systems do not provide the power and flexibility sufficient to manage the task of IC manufacturing. An SPR 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 and processing history. An SPR must have the extensive power of a full-function programming language.

1.4 Semiconductor Process Representations

Several SPRs have been developed that attempt to overcome the limitations of current commercial systems. These SPRs are described and compared with our approach in this section.

There are two basic approaches to the design of an SPR: knowledge-based and procedural. A knowledge-based approach uses a hierarchical, object-oriented data structure to represent a process flow. 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 which contains attributes or slots that specify the parameters of the operation. A method is defined on the class that defines the semantics of the operation. The interpreter for a knowledge-based SPR is implemented by writing a program that traverses the data structure that specifies a process flow and calls the appropriate method. The method defines appropriate semantics for the operation.

The advantage of the knowledge-based approach is that new operations can be defined as a subclass of an existing class so that default parameters can be inherited from an existing operation.

Other advantages are that the data structure can be conveniently stored in a relational database and it is relatively easy to write programs that operate on a process flow because it is just a data structure. It is also relatively easy to make certain modifications to the process flow, such as inserting a sequence of steps.

The disadvantage of this 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 an operation is violated).[1] The knowledge-based representation emphasizes the correct behavior of the process.

A procedural, or programming-language, approach represents a process flow by a program. A procedural process flow is defined by 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 [17] which is roughly equivalent to the data structure in a knowledge-based representation. The interpreter for a procedural SPR is similar to the interpreter in the knowledge-based approach. However, it implements a full-function programming language rather than a limited set of primitive operations. 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 that 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. Both approaches have essentially the same expressive power. Consequently, the kind of representation is less important than the particular constructs and abstractions that are provided.

Before proceeding to consider other SPRs a brief introduction to the SPR used in this dissertation will be presented. BPFL is a programming-language approach to SPR. A simple example of a BPFL specification appears in Figure 1-6. The figure contains an outline of the BPFL descrip-

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

```
defflow anneal-implant(time: = {30 min}, temperature: = {950 degC})
  " Anneal implant damage in nitrogen ambient "
let segments := find-segments-in-lot(material: #m(substrate));
begin
  viewcase
    when simulation do
      ...
    end;
    when fabrication do
      with-equipment furnace of-type 'n2-furnace do
        run-recipe(furnace, 'anneal, time: time, temperature: temperature);
      end;
    end;
    segment-material-attribute-in-lot(segments, :implant-annealed) := t;
  end;
end;
```

Figure 1-6: BPFL specification example.

tion for a process step called `anneal-implant`. This is a standard operation for repairing substrate damage caused by ion implantation. The step has two arguments: `time` and `temperature`. A sample of BPFL code using the step is:

```
n2-anneal(time: {1 hr});
```

In this case, the `time` argument is set to 1 hour using the units data type (i.e., magnitude and unit designation enclosed in set braces) to denote dimensioned quantities. The `temperature` argument is not supplied in the above example and so it takes on the default value of 950 °C specified in the procedure definition.

BPFL uses a *view* mechanism to specify information that is of interest to selected interpreters. For example, code for simulators appears in the `simulation` view and code for the WIP interpreter is in the `fabrication` view. In addition, BPFL programs maintain a model of wafer state that is used to check processes for correctness, to store measurements, and to permit the movement of wafers between different runs. The line of code:

```
segment-material-attribute-in-lot(segments, :implant-annealed) := t;
```

updates the wafer-state model to indicate that the wafers have been annealed.

BPFL has statements to specify control flow, (e.g., `if-then`) and common abstractions encountered in processing (e.g., rework loops, timing constraints, etc.). More detail is given in later chapters.

The earliest attempt to develop a formal process-flow representation was FABLE which is a procedural SPR [18],[19]. FABLE programs are structured in terms of a hierarchy of *layers* cor-

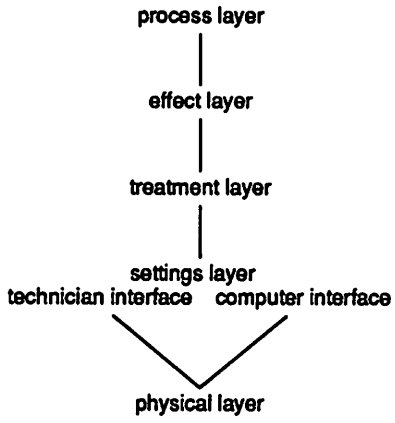

Figure 1-7: Fable standard layer hierarchy.

responding to the level of abstraction of an operation. The layer hierarchy is shown in Figure 1-7. The *process* layer represents fabrication operations (e.g., grow-gate-oxide). The *effect* layer corresponds to operations resulting in a change in the material being processed (e.g., growing an oxide layer). The *treatment* layer corresponds to operations performed on wafers to achieve the desired effect. The *settings* layer describes equipment settings used to achieve a desired treatment. It has *technician* and *computer* interfaces that describe the operation of equipment in terms appropriate for use by operator interaction and automatic equipment control, respectively. For example, the technician interface for a furnace includes the required values of recipe parameters and recipes suitable for input on the furnace front panel, and the computer interface view includes the equipment parameters suitable for downloading to the furnace controller. The lowest-level view is the *physical* layer, which indicates the appropriate actions to take to turn settings into physical reality (e.g., a setting of 900 °C can be achieved by setting the temperature control appropriately).

Higher-level layers are implemented in terms of lower-level layers. For example, a grow-oxide operation defined at the effects layer is implemented in terms of furnace settings defined at the settings layer. Layers are only permitted to refer to layers of a lower-level of abstraction.

FABLE programs are written in terms of *specifications* describing each layer and *implementations* describing the steps to take to move between layers. Figure 1-8 presents a FABLE definition for diffusion, for both the treatment layer and the treatment-settings interface.

FABLE has not been successful. The strict hierarchy, while attractive for compartmentalizing the specification of equipment and processes, is too restrictive for use in a real manufacturing

```
Diffusion-Treatments = subclass of Treatment-Library with
  specification of [Treatment layer] =
    Slow-Push:Operation(wafers: Lot; temp: Temp; environment: Gases);
    Diffuse:Operation(wafers: Lot; temp: Temp; duration: Time; ...);
    Slow-Pull:Operation(wafers: Lot; temp: Temp; environment: Gases);
end [Treatment layer];

implementation of [Treatment layer] =
begin
  for Furnace use [Technician layer];
in
  Diffuse =
    begin
      f := wafers.station ! The furnace the wafers are in
      temp-setting := Calculate-Temp-Setting(temp,f);
      f.Set-Temp(temp-settings);
      ...
    end Diffuse;
end [Treatment layer];
```

Figure 1-8: FABLE specification example.

environment. Furthermore, FABLE was not designed to be used for applications other than a WIP system so it makes no allowances for the use of process simulators. It has a limited ability to perform safety checks on fabrication operations but it has no notion of wafer state so it cannot base such checks on the past history of a lot unless the programmer codes it explicitly. It also does not adequately address the real-time issues encountered in semiconductor processing.

The Process Flow Representation (PFR) is a knowledge-based approach developed at MIT [20]. PFR is based on a three-level process model shown in Figure 1-9 that is similar to the FABLE layer hierarchy. The focus of this model is the transformations wafers undergo during processing. Each operation performed on a wafer is modelled in terms of three levels of information appropriate

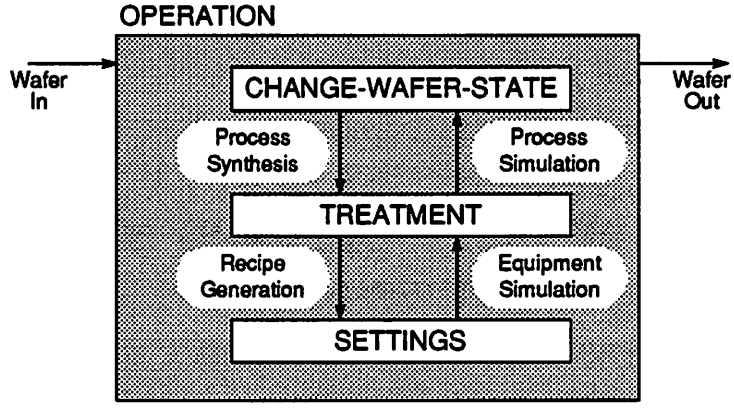

Figure 1-9: Three-level process model.

to a particular domain. The *change-wafer-state* level expresses information about the change a wafer undergoes during an operation. The *treatment* level describes the physical environment around the wafer causing the change (e.g., temperature and gas flow in a furnace). The *settings* level describes the parameters for an equipment recipe needed to achieve the desired treatment. This model is also known as a two-stage model in reference to the translations between levels rather than the levels themselves.

An example of a PFR specification for a furnace operation is shown in Figure 1-10. The basic unit of processing is an *operation*, which has attributes `:change-wafer-state`, `:treatment`, and `:settings` which correspond to the levels in the basic model. Additional attributes can also be used, such as `:time-required`. The PFR uses these attributes to capture a declarative description of an operation which is built up hierarchically to form complete process flows.

The PFR specification is translated into an instance hierarchy in the object-oriented database Gestalt [7]. Each instance in the database has slots corresponding to those present in the PFR process flow. As with other knowledge-based SPRs, a process flow is executed by traversing the object tree that defines it, and acting on the values contained in the slots relevant to the task being performed. For example, a scheduler may only be interested in the values present in the `:time-required` slot. This approach differs from the views approach used in BPFL. An attribute in PFR

```
(define stress-relief-oxide
  (operation
    (:documentation
      "Stress Relief Oxide to minimize stress effects of nitride deposition")
    (:time-required (:mean (:hours 7 :minutes 15) :range (:minutes 5)))
    (:body
      (operation
        (:permissible-delay :minimal)
        (:body
         rca-clean
         (operation
            (:change-wafer-state
            (:oxidation :thickness (:angstroms (:mean 430 :range 20))))
          (:treatment
            (furnace-rampup-treatment :final-temperature ...)
            ...)
          (:settings (:machine GateOxTube :recipe 210)))))
      ...)))
```

Figure 1-10: PFR specification example.

is a basic piece of information about a process, whereas a view consists of many attributes designed to support a particular activity (e.g., the scheduling view in BPFL).

PFR has limited support for control-flow in processes. It has an `if` operation but does not currently support loop operations, lot splits and merges and exception handling.

The Manufacturing Knowledge System (MKS) developed at Schlumberger [21] is another knowledge-based approach to SPR. MKS is built on an object-oriented programming environment, named Hyperclass.[1] Process flows are represented in terms of steps, which are considered to have three basic components: a step body, a set of input ports (*inports*) and a set of output ports (*outports*). The step body describes the intended function of the step, including specific parameters (e.g., temperature in an oxidation step), preconditions that must exist before the step may be executed (e.g., wafer must be cleaned before being oxidized), and an input-output transformation that describes what is supposed to happen within the step (e.g., silicon surface will be consumed and an oxide surface will be added during a thermal oxidation step). Inports define conditions for wafers entering the process step (e.g., an inport might require that entering wafers must have been cleaned within the last thirty minutes). Similarly, outports define conditions for entities (e.g., wafers, materials or status information) leaving the process step (e.g., an output might specify that exiting wafers are covered in an oxide layer of a certain thickness). All steps are divided into a hierarchy and inherit attributes from their parents.

Complete processes are created by "wiring together" instances of existing process steps. A unidirectional link is created from an output of one step to an inport of another. Similarly, each inport of the newly-defined high-level process is connected through a special link, called a *correspondence link*, to an inport of one of the steps contained within it. This linking establishes an equivalence whereby entities entering an inport of the high-level process are considered to be entering a corresponding inport in a lower-level substep.

An example of an MKS specification of the steps required to prepare a wafer for spin-coating with resist is shown in Figure 1-11. There is an implicit rework loop between the `particle-inspection` and `wafer-scrub` steps. MKS recognizes three step types. First, *processing* steps which transform the material being processed. Ports in processing steps are `wafer-in` and `wa-`

[1] Hyperclass is a trademark of Schlumberger Technologies Inc.

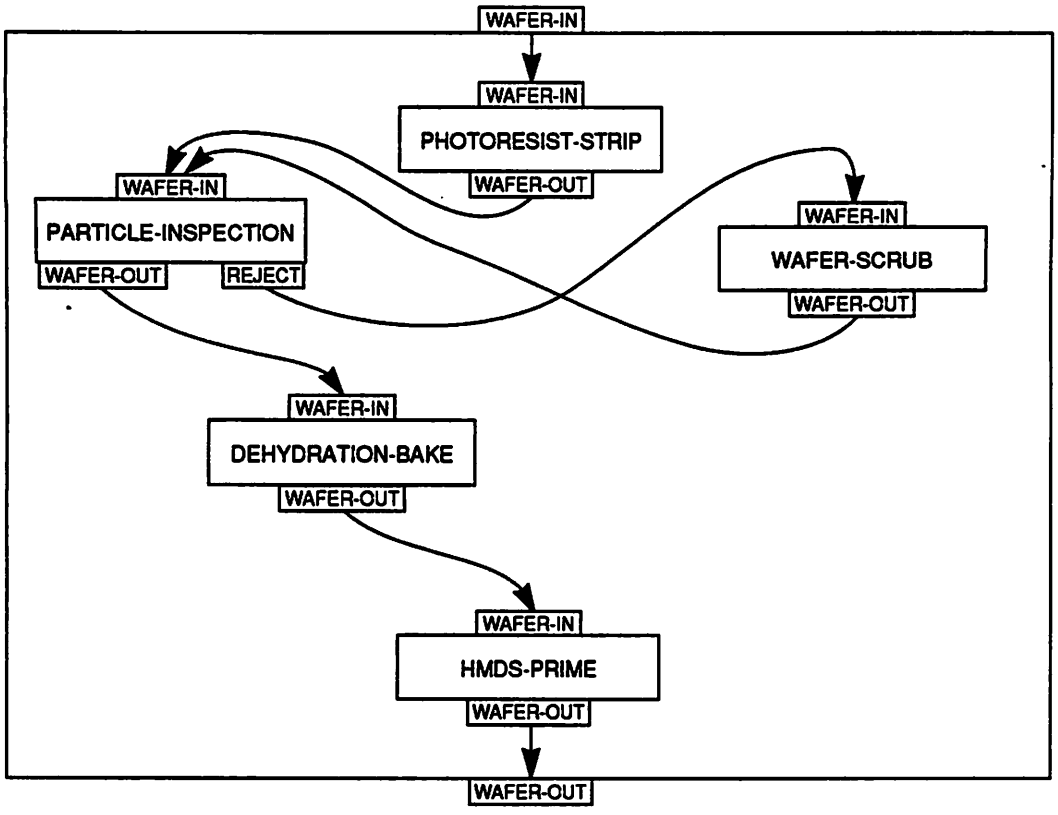

Figure 1-11: MKS process specification example.

`fer-out` only, as seen in the `photoresist-strip` step in Figure 1-11. Second, *testing/measurement* steps which produce information as their primary output, either explicitly or attached to wafers leaving the step. Third, *decision* steps sort incoming wafers according to results from previous testing steps. The `particle-inspection` step in the figure is an example of a composite step, consisting of a testing step followed by a decision step. The step has two outports, `wafer-out` and `reject`, and wafers are sorted and transferred to an outport based on the result of a particle inspection test.

MKS has been used to implement the AESOP diagnostic system [22]. Diagnostic knowledge is represented as a network of causal links that associate effects (e.g., oxide too thick) with causes (e.g., anomalous temperature reading). Given test data, AESOP employs diagnostic inferencing to isolate possible causes of failure and a relative measure of their likelihoods. MKS has also been used to develop a SUPREM simulator interface.

MKS is well-suited for use in process design and simulation; however, it lacks the exception-handling mechanisms and timing constraints required for use in fabrication.

Another knowledge-based approach to SPR is used in the Process Design Aid (PDA) developed at Stanford University [23]. PDA simplifies process synthesis by consolidating the use of simulation tools. PDA process flows consist of a hierarchical structure of process steps in which each step is a class instance with inheritance and defaulting of attributes. PDA permits extensive use of libraries of simulation results and allows for tuning a process based on simulation output and vice-versa [24]. It is possible to iteratively improve process and simulation parameters with the aim of achieving greater concordance between simulated and fabricated devices. PDA emphasizes the careful tuning of existing process steps and tends to view processes from the bottom up, with finely-crafted lower-level steps.

PDA differs from other knowledge-based approaches in that it makes extensive use of the *prototype-instance* objects provided by HyperClass. In this system, each instance of a class itself forms a unique class which can be specialized. Most object systems are class-instance systems, in which classes may be redefined but instances may not be specialized beyond their class definitions. Inheritance in PDA occurs along two hierarchies, known as the *is-a* hierarchy and the *part-of* hierarchy. The is-a hierarchy corresponds to the *class-of* relation in a conventional object system. The part-of hierarchy corresponds to object decomposition and permits specialization of instances, so users are free to add attributes in any way they desire. This makes the system more flexible than other knowledge-based systems, but introduces the problem that user attributes may not be interpretable by other PDA tools. PDA has been integrated into the MKS and works with the tools developed for MKS.

While PDA's interface to simulation is probably the best of the systems discussed here, it too lacks many of the fabrication-specific statements that BPFL possesses. There is no support for exception-handling or timing constraints.

Another approach to SPR is a graphical language being developed at Siemens to express process flows for a WIP system [25]. In this approach process flows are represented as trees of parameterized subprocess plans. The leaves of the trees correspond to actions that can be carried out directly. Figure 1-12 shows an example of a plan tree for a process flow that is composed of two

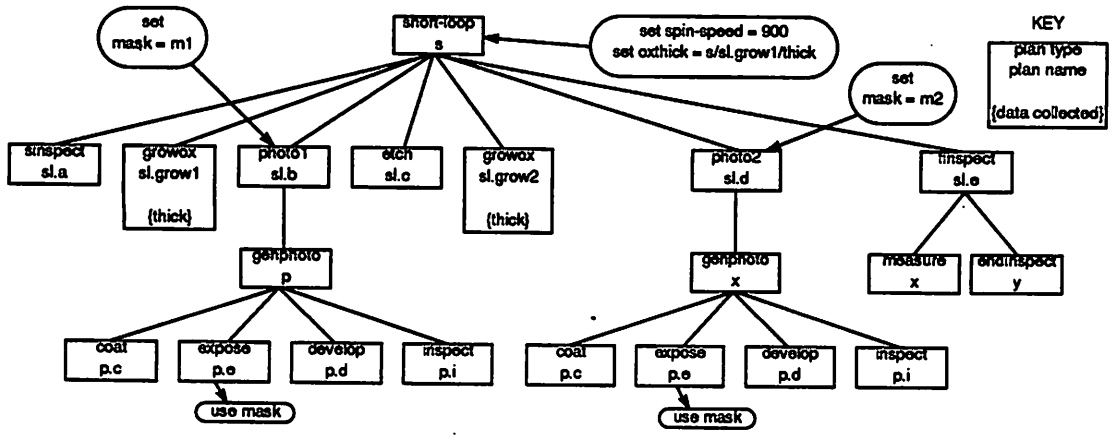

Figure 1-12: Siemens tracker process flow example.

oxidation and two photolithography steps. Each step (or *plan*) has associated with it a type and a name. Additional attributes may be attached (e.g., comments may be added, and indicators may be used to specify that certain data should be collected when a step is executed). For example, the `growox` step calls for the measurement of oxide thickness. Parameters are supplied by defining values used by the atomic processing operations found at the leaves (e.g, the variable `mask` is assigned the value `m1` in the `sl.b` step which is used in the `expose` operation).

The advantage of a specification like that in Figure 1-12 is that it is very easy to store in a conventional relational database and may be executed by performing a depth-first traversal of the plan tree. Since the process-flow interpreter runs as a database application it is very robust. However, the language does not have the power of a general-purpose programming language (e.g., it is not possible to construct conditional statements), and it is only intended for use in a WIP tracking system. It also has no mechanism for dealing with exceptions, timing constraints or wafer state checks.

Siemens is also working on a procedural SPR called the Manufacturing Programming Language, version 2 (MPL.2) [26]. An example of an MPL.2 process flow is shown in Figure 1-13. MPL.2 is intended for use with Intelligent Migrating Processes (IMPs). An IMP is a process that explicitly migrates between computers. When the lot moves to another workcell, the IMP for the lot is moved to the local computer of that workcell. IMPs can move to different computers and operating systems. An interpreter for IMP programs exists on each machine. Migration between machines is supported by stopping an IMP on its current machine, encoding the execution state of the

```
...
moveto(oneof("wetsink"));
execute("surface-prep"));

redo = 0; maxredo = 5; result = "bad";
loop
  while (redo <= maxredo) do
    moveto("coater1");
    execute("coat+bake");
    ...
endloop;

if (redo > maxredo) then
  scrap()
endif;
...
```

Figure 1-13: MPL.2 example.

IMP into an external representation that is sent with the code to the target machine, decoding the state into a format suitable for the new machine, and resuming execution at the statement following the migration request. Migration to a new machine is specified by the **moveto** statement as illustrated in the figure. While resident on a machine, an IMP may access any of the local resources of that machine (e.g., the **execute** statement runs a recipe stored at the local machine). Access to remote resources is also available through interprocess communication.

MPL.2 is a full-function programming language so complex control flow for operations such as rework and control loops are easy to implement. The disadvantage is that IMPs are only suitable for WIP tracking. While the idea of moving an executing SPR with the lots it controls is novel, it is not apparent that this approach has any significant advantage over the distributed heterogenous DMBS-based system using cell controllers outlined in section 1.2.

The desire to check processes for correctness before they are used in production has driven Hitachi to develop a process-flow validation system as part of their Laboratory Automation (LA) project [27]. The Hitachi system uses a rules-based expert system for process flow debugging and validation. The system is designed to point out incorrect or questionable conditions in a flow that could have undesirable effects on wafers or equipment. Process knowledge is grouped into four types:

1. Process window – Upper and lower limits and allowed conditions (e.g., furnace temperature or gas species allowed in a machine).

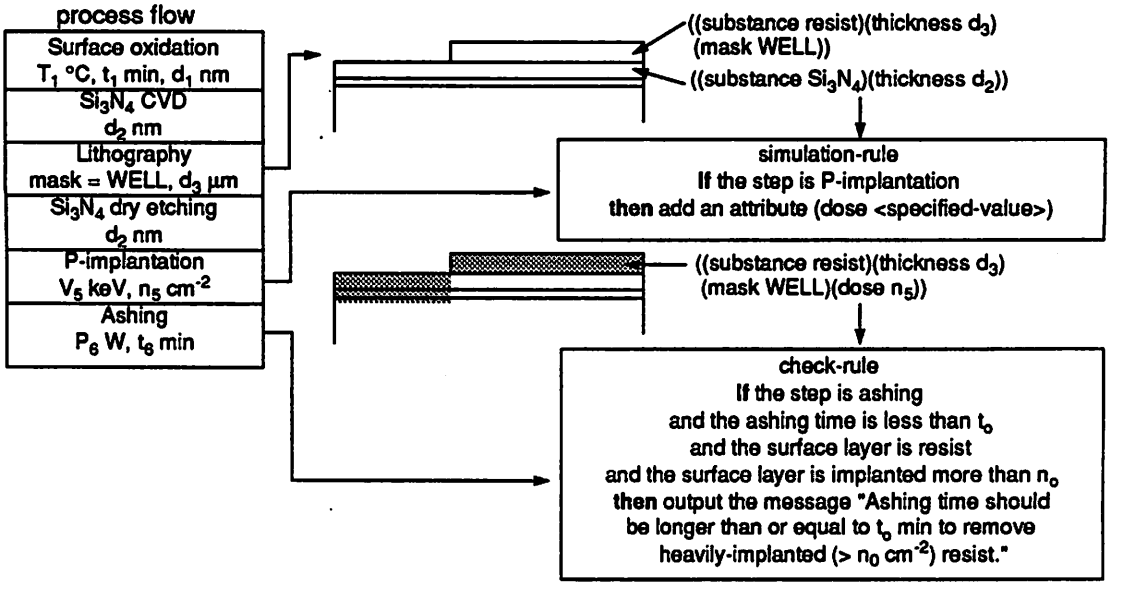

Figure 1-14: Hitachi process flow and check rules example.

2. Process sequence – Acceptable sequences of operations in a process (e.g., a furnace operation must be preceded by a clean).
3. Wafer-process constraint - Constraints between wafer state and process or equipment (e.g., contaminated wafers should not be loaded into clean furnaces).
4. Optimum condition – Optimum conditions to fabricate intended structures or characteristics (e.g., best gas flow rate and temperature for low-stress nitride deposition).

The first three types of knowledge result in rules that are simple to express and that must be met in a process for it to work correctly and to avoid equipment damage. The fourth condition concerns processing at suboptimal conditions. While this is undesirable because it reduces product quality, it does not result in damage to wafers or equipment. In order to check the process for correctness, the system maintains a wafer state description similar to that used in BPFL.

An example of process checking is shown in Figure 1-14. The process flow being checked is shown on the left of the wafer profile. At the `lithography` step, the wafer schematic is as illustrated in the upper right of the diagram (only the properties of the `resist` and Si_3N_4 regions of the wafer are shown in the illustration). Each region of the wafer has a series of attribute/value

pairs describing the state. Once the `P-implantation` step is reached, a simulation rule fires that instructs the system to add a `dose` attribute to the exposed regions of the wafer, as reflected in the state shown in the lower wafer schematic. When the `ashing` step of the process flow is reached, a process check rule (in this case, a wafer-process constraint relating the knowledge that heavily-implanted resists are difficult to remove) fires and outputs a warning message if $t_6 < t_0$ and $n_5 > n_0$.

The SPR used in this project is not mentioned in the literature, but it is probably similar to the representations used in COMETS and WORKSTREAM. However, the system is noteworthy because it illustrates the importance of process checking and the need to maintain wafer state. Hitachi claim that the process checking system has halved process design time.

1.5 Process Specification in other Industries

This section briefly describes how other industries have dealt with the process specification problem.

Most industries have focussed on product definition languages with the intention of automating fabrication based on part descriptions, rather than process-flow languages. This approach works well in metal parts machining, for example, where flow is almost always sequential (i.e., no rework) once the task of generating the process plan is complete. One example of a process plan generator is GARI [28]. GARI is given the design of a mechanical part, and it produces a plan to machine the part based on rules expressing technological limitations and economic considerations, such as:

1. If a hole H2 opens onto another hole H1, then machine H1 before H2 (to avoid damaging the drill),
2. If several operations must be performed on the same machine then try to group the operations (to reduce cost).

GARI is based on an expert system. Later efforts have linked process plan generators to workcell programming languages to simplify product testing and improve operator interactions [29].

Process plan generation is of almost no value in the semiconductor industry because it is very rare that operation order may be altered from that specified by process designers. Automatic recipe generation for processing equipment is an area of active research [30] but is used to optimize the processing conditions of a given operation and not to alter the order of operations in the process

```
PARTNO XC CIRCLES
MACHIN/GN5CC,9,OPTION,2,0
CLPRNT
CUTTER/0.5
FEDRAT/2.0
TOLER/0.0005
P0=POINT/0,0,0
C1=CIRCLE/3,0,0,2
L1=LINE/(3,0,0),(3,1,0)
FROM/P0
GO/C1
AUTOPS
TLLFT,GOLFT/C1,ON,L1
END
FINI
```

Figure 1-15: APT program example.

flow. Furthermore, recent automatic plan generation projects have run into complexity problems [31]. As parts become more complex, two basic problems emerge:

1. the combinatorial explosion of the number of possible process plans, and
2. the need to have local knowledge of machining capabilities, dependencies and the availability of tools, toolblocks and fixtures.

One area where process-flow languages have survived is in the area of numerically controlled (NC) machines. The dominant language for NC machines is APT (Automatically Programmed Tool) [32] which evolved from a 1955 effort to computerize machine control into a language with process description capability. Figure 1-15 shows an APT program that defines a circular contour cut. APT's age is apparent from the code in the figure. An APT program may be run on many machines because it is run through an APT postprocessor, which generates code for a particular type of machine. APT programs consists of four basic statement types:

1. Geometric statements that define the part configuration (e.g., lines, planes, holes).
2. Motion commands to control the path of a cutter.
3. Postprocessor commands which control different machine functions (e.g., spindle speed, feedrate).
4. Language control instructions which generate geometric translations, rotations and diagnostics.

The major problem with APT is that it is an extremely domain-specific language, suited only for

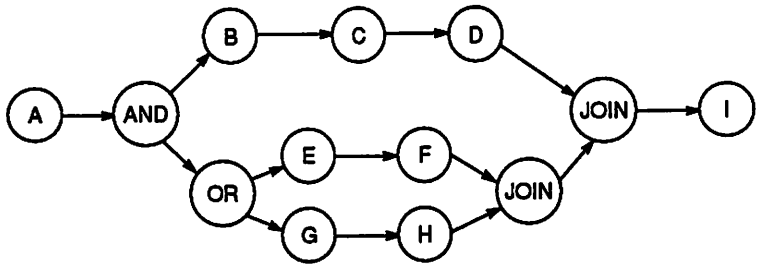

Figure 1-16: ALPS specification example.

NC machine use. The requirement for an APT post-processor for each type of machine is a major drawback. APT contains no constructs to specify complex, parallel processes and is intended to describe only the actions of an isolated machine. Finally, by today's standards APT has poor syntax, does not have sophisticated abstraction mechanisms, and is difficult to read. It is very poorly suited for use as an SPR.

Recently the problems of optimizing machine usage, particularly with industrial robots, have been studied. These efforts appear to be overcoming many of the shortcomings of APT, since they employ more advanced software engineering technology and they deal with multiple machines [33]. However, they remain very domain specific.

The proliferation of domain-specific languages and the diversity of controllers in manufacturing (e.g., NC machines, robots, programmable logic controllers, and materials handling systems) has resulted in efforts to develop systems to manage the entire manufacturing operation [34],[35]. As part of this effort, the National Institute of Standards and Technology (NIST) has developed a language with somewhat similar goals to the SPR's outlined in the previous section. ALPS (A Language for Process Specification) [36] is intended principally as an interface between process planning and production control. During production it is also used to drive production control processes. ALPS notation is a directed graph, as shown in Figure 1-16. The language provides extensive control over processing. It allows for alternative sequences and parallel actions which none of the SPRs considered here can express. ALPS also provides for synchronization between parallel tasks and between multiple processes. While ALPS has much to recommend it in terms of power and flexibility, it should be noted that at this point ALPS is little more than a prototype language design and has not yet been implemented. Since ALPS is intended to be useful in a wide range of industries,

no detailed syntax beyond the top-level graph notation has been devised. A full-scale implementation of ALPS will have to await the deployment of a prototype for the manufacturing system control software in which it is intended to operate [35].

Automating semiconductor manufacturing is especially difficult. Processes in the semiconductor industry are complex and poorly understood. Equipment used in semiconductor manufacturing tends to be expensive, complex, and unreliable, requiring highly-skilled operators and technicians. The very short process and product lifetimes mean that typical semiconductor manufacturing equipment is never fully debugged. For example, in 10 years commodity DRAM density has gone from 16K to 1M, which required 4 generations of processes each requiring substantially more powerful and complex equipment than its predecessor. This situation is particularly true with respect to machine contamination and long-term reliability. Many pieces of wafer fabrication equipment are sufficiently specialized that fewer than 100 machines of a given type are ever constructed. Such small number of equipment instances makes vendors reluctant to invest in equipment automation. Consequently, standards for equipment automation are less developed than in most industries.

Another problem is that the cost of clean room space and equipment means that fabs are often shared by different processes with their own requirements. The need for extreme cleanliness both in terms of particulates and potential cross-contamination accentuates these problems in shared facilities.

In summary, the dynamic nature of semiconductor processing places much greater demands on automation than in most industries. Another difference between products in a continuous process, such as the semiconductor industry, and other discrete-part industries is that integrated circuits are monolithic. In discrete-part industries complex parts are composed of many subcomponents which are assembled to make the final product. For example, a Boeing 747 contains several million parts, but in the fabrication process, progressively more complex subassemblies are constructed until final assembly involves only a few major components. Failure in any one part can usually be repaired simply by replacing that part. In contrast, semiconductor fabrication involves just one component (a silicon wafer) which is subjected to a large number of treatments. The resulting wafer is divided up into individual circuits (*dies*) near the end of fabrication. Failure of any one

device on a wafer typically results in the loss of one complete die. An abnormality in one of the treatments the wafer undergoes typically results in the loss of the entire wafer (often the entire lot of wafers).

For these reasons, research into process plan generation and process-flow languages in other industries are difficult to apply to the semiconductor industry.

1.6 Summary and Dissertation Outline

This chapter has described the need for a CIM system and the importance of a powerful SPR. Commercially-available CIM systems have been reviewed and it has been shown that their shortcomings are too severe to permit the implementation of powerful operations such as control loops, timing constraints and exception handling.

The Berkeley approach to CIM has been presented. It is based on a distributed computer system with multiple levels of control and distributed databases. The system envisaged should be flexible enough to cope with the demands of an advanced CIM system.

Two basic approaches to the design of an SPR were discussed: procedural (e.g., FABLE and BPFL) and knowledge-based (e.g., PFR, MKS, PDA). Each approach has its advantages. Knowledge-based SPRs are generally easier to maintain and update, but they lack the programming features (e.g., exception-handling) provided by procedural SPRs.

Process specification techniques in industries other than IC manufacturing were examined. Most work in process representation has been with process description languages, where the intent is to automatically generate process plans based on part descriptions. Primitive process-flow languages are used in NC machine programming. The ALPS language being developed by the NIST looks promising, but is currently no more than a paper design without an implementation.

This dissertation describes BPFL and a WIP system based on it. Chapter 2 describes BPFL. Chapter 3 describes statements in BPFL intended primarily for use in fabrication. Chapter 4 discusses the WIP system, focussing on the run-management system. Chapter 5 describes the implementation of the WIP system. Lastly, chapter 6 presents conclusions and directions for further research.

[This page intentionally blank]

Chapter 2
The Berkeley Process-Flow Language and Interpreters

This chapter describes the structure of a BPFL process flow and abstractions supported by the language developed to support semiconductor manufacturing. First, the design goals and assumptions that led to BPFL are outlined. Second, the syntax of BPFL and some basic operations are introduced. Third, BPFL equipment abstractions are described. Fourth, the BPFL wafer-state representation is discussed. Finally, the database schema designed to represent BPFL programs in a database are described. BPFL is described by showing examples of a standard CMOS process flow.

2.1 The BPFL Approach to Process Specification

BPFL is a procedural SPR. The language is designed to allow all information about a process to be merged into a common specification. Different programs, called *interpreters*, execute BPFL programs and perform specific tasks. For example, a WIP interpreter executes a BPFL program and issues instructions to equipment or operators to carry out the necessary steps to fabricate the product described by the program. A simulation interpreter executes the program and generates input for simulators that can be run to predict the performance of fabricated devices [37]. Other interpreters can be implemented to perform different tasks such as factory simulation.

This approach minimizes the amount of domain-specific knowledge about particular tasks required by BPFL. For example, if BPFL were capable of describing input sufficient for use with any simulator, there would be tremendous language overhead in coping with the numerical switches and environment settings for device simulators like PISCES [38]. However, an interpreter written to generate input for PISCES (and perhaps other simulators as well) can deal with these details. There are situations where a BPFL program has to include such information for the use of a particular simulator or to work around interpreter bugs, but the aim is to minimize the need for this information.

The design goals for BPFL are to:

1. provide a common specification suitable for use in all stages of manufacturing,
2. provide support for a complete specification including lot splits and merges,

exception handling, timing constraints, rework loops, feed-forward and feedback control, and equipment communication, and

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

Knowledge-based approaches are generally weak at handling exceptions and timing constraints. We believe these features are essential in any SPR intended for use in an environment where the potential for unexpected errors and mishaps is great.

This chapter describes the BPFL language in detail, including a discussion of the interpreters and some advanced language concepts. Additional information about the features of BPFL intended to support fabrication are discussed in chapter 3. A complete specification for BPFL is given in Appendix A.

2.2 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 [39]. Lisp was chosen as the host language for several reasons. First, it is easy to develop programs that manipulate other Lisp programs since they are represented using list data structures that can be accessed from Lisp. BPFL interpreters operate on BPFL programs so using Lisp simplified their development. Second, Lisp provides a very powerful and flexible framework within which to experiment with language designs. The current version of BPFL is quite different from the original version [40] and Lisp greatly reduced the amount of work necessary to make these changes. Furthermore, Lisp has a built-in evaluator that makes it very easy to implement language interpreters [41]. Lastly, a well-defined and powerful object-oriented programming model, the Common Lisp Object System (CLOS), was already available. CLOS is used extensively both in the design of the language and in the implementation of the interpreters.

Although BPFL is a Lisp-based language, the language seen by users is not Lisp. We intend to provide a user-friendly, forms-based, graphical user-interface. Examples of such interfaces are the graphical representation of process-flows in the MKS system (see Figure 1-11) [21], the exper-

```
defflow cmos-16(implant-split: = t)
  "U.C. Berkeley Generic CMOS Process (Ver. 1.6 14-April-89)
   (2 um, N-well, single poly-Si, single metal)"
begin
  step ALLOCATE-WAFERS do ...;
  step WELL-FORMATION do ...;
  ...
end;
```

Figure 2-1: Berkeley `cmos-16` process flow in BPFL.

imental user-interface to the process-flow representation developed at Texas Instruments [42], and the Stanford Graphical Design Toolkit [43].

Currently, users are presented with a block-structured textual language for BPFL. An early version of BPFL used a Lisp syntax which the intended users (i.e., process engineers) found unsatisfactory. Also, Lisp syntax is difficult to read. The block-structured BPFL is easy to translate to the Lisp syntax and vice-versa.

A process-flow is represented by a BPFL procedure that contains a sequence of steps. Each step contains a sequence of BPFL procedure calls, BPFL statements, and Common Lisp function calls. Figure 2-1 shows the top-level of the standard CMOS process that is run in the U.C. Berkeley microlab. BPFL procedures are defined using the **`defflow`** definition. This definition has four arguments: the procedure name (e.g., `cmos-16`), a formal argument list (`implant-split: = t`), an optional documentation string (`"U.C. Berkeley Generic CMOS..."`), and a procedure body (**`begin step`** `ALLOCATE-WAFERS ...` **`end`**). The procedure body contains a sequence of process steps. The Berkeley `cmos-16` process-flow has sixteen top-level steps. The first step allocates the wafers that will be processed by the run, the second step creates an n-doped well for the PMOS devices, and so forth.

`Step` is used primarily for documentation purposes. The first argument to the **`step`** statement is a symbol that names the step. Since steps can be nested, the names of all current steps concatenated together is used to indicate the position at which an event occurred. This position name is called a *step-path*. The step-path is used to identify specific statements in a program when recording information about a run. For example, the step path is recorded whenever a measurement is recorded.

BPFL provides abstractions to manipulate wafers and lots, since they form the basic units on which processing is performed. Wafers are represented by CLOS objects, each with a unique identifier (called a *wafer ID*) that distinguishes it from all other wafers.[1] The identifier inscribed onto the wafer is recorded, and the wafer is assigned a logical number index which is used to identify it within a run. Wafer indices are integers between 1 and the number of wafers being processed by the run. Wafer objects also have a data structure associated with them for storing wafer state information which is described in more detail in section 2.4.

A *lot* is a named set of wafers. Predefined lot names are supplied for wafers that are intended for production (`product`), wafers that are to be scrapped (`scrap`) and wafers that need rework (`rework`)[2]. There is also a lot that contains the wafers currently being operated on (`current`). A given wafer may be in several lots at the same time, with the exception that wafers in the `scrap` lot may appear only in that lot. BPFL programs can define new lots to hold test wafers or identify subsets that will receive special processing.

Procedures are provided to create (i.e., initiate) and destroy (i.e., terminate) lots and to add 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 pre-existing lot, or by starting a new run and passing it a set of wafers. Procedures are also provided to merge lots.

Examples of the use of these lot operations are shown in Figure 2-2. This is the BPFL representation of the first two steps in the CMOS-16 process flow shown in Figure 1-5. The first step allocates wafers. The second step is an initial oxidation that grows an oxide mask used later in the well definition.

In BPFL, 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, which creates a specification of the wafers to be allocated, uses argument passing by name for four arguments: the desired crystal orientation of the wafer (`crystal-face`), the desired resistivity (`resistivity`), the wafer

---

[1] Industrial efforts to establish a common bar-coding convention for wafers will provide every manufactured wafer with a unique id.

[2] Since rework operations may nest, rework lots actually take the names `rework-1`, `rework-2`, etc.

```
step ALLOCATE-WAFERS do
  let spec := bare-silicon-wafer(crystal-face: 100,
                                 resistivity: [{18 ohm-cm},{22 ohm-cm}]
                                 quality: 'product, dope: 'p);
  begin
    allocate-lot(names: '(cmos, nwell, nch),
                 sizes: list(*product-lot-size*, 1, 1),
                 snapshot: spec);
  end;
  /* Wafers in the cmos lot are product wafers */
  lot('product) := lot('cmos);
  with-lot 'nwell do
      measure-bulk-resistivity(tag: "initial");
  end;
end;

with-lot 'cmos do
  step WELL-FORMATION do
    step INIT-OX do
      wet-oxidation(time: {11 min}, temperature: {1000 degC},
                    target-thickness: {1000 angstrom});
      pattern(mask-name: 'NWELL);
    end;
  ...
end;
```

Figure 2-2: BPFL representation of `cmos-16` initial steps.

quality (`quality`), and the type of background dopant (`dope`). The supplied values mirror those given in Figure 1-5.

The resistivity argument uses a range value. Range values are denoted by using square brackets. For example, the construct `[1,10]` represents the range 1–10. BPFL also supports dimensioned quantities (i.e., values with unit designators) through the use of set brackets. For example, "1 cm" may be represented as `{1 cm}`. The `resistivity` argument specifies a wafer resistivity between 18–22 ohm-cm.

The next operation creates wafers and assigns them to lots. The `name` argument specifies the names of the lots, and the `size` argument specifies how many wafers to allocate. In this case, three lots named `cmos`, `nwell`, and `nch` with sizes `*product-lot-size*`[1], `1` and `1` respectively are allocated. The `snapshot` argument specifies the wafer state. Since these are new wafers, the wafer specification is the result returned by the `bare-silicon-wafer` procedure. The fol-

[1] `*product-lot-size*` is a global constant that contains the number of product wafers to allocate for processing in this run.

```
let temp := lot('current);
begin
  lot('current) := lot(lot-names);
  operations;
  lot('current) := temp;
end;
```

Figure 2-3: With-lot semantics.

lowing operation sets the product lot to the cmos lot so that the interpreter knows which wafers are product wafers and which are test wafers.

The **with-lot** statement is used to indicate which lots of wafers are to be treated as current. With-lot changes the value of current for those operations within it. After the **with-lot** is complete, current assumes the value it had before. For example, the semantics of:

```
with-lot lot-names do
  operations;
end;
```

are shown in Figure 2-3. The only operation in the **with-lot** statement is measure-bulk-resistivity. Because the current lot has been set to nwell, this procedure will operate on the wafer in the nwell lot. This is the final operation in the ALLOCATE-WAFERS step.

The next operation starts the WELL-FORMATION step. The INIT-OX step (corresponding to step 2 of Figure 1-5) is nested within the WELL-FORMATION step. 2.1–2.3 in Figure 1-5 are replaced by a call on wet-oxidation, which is a standard library procedure that performs the necessary wafer and furnace cleans prior to oxidation, and measures the oxide thickness after oxidation. Only rarely is an oxidation performed without all three of these operations being required so it makes sense to define one procedure to do all of them. Step 2 of Figure 1-5 includes specification of the required pre-oxidation, post-oxidation, and anneal times whereas the BPFL specification in Figure 2-2 does not. This serves as an example of default parameters as described in the definition of the wet-oxidation procedure shown in Figure 2-4. There are seven named arguments: time, temperature, pre-ox-time, post-ox-time, anneal-time, target-thickness and tag. All except time, target-thickness and tag have default values. In the case of the procedure call in Figure 2-2, the temperature default is overridden but the other defaults are used. Default parameters simplify process flows. They provide a mechanism to hide detail while permitting specification of detail if necessary.

```
defflow wet-oxidation(time:, temperature: = {900 degC},
                      pre-ox-time: = {5 min}, post-ox-time: = {5 min},
                      anneal-time: = {20 min}, target-thickness:,
                      tag:)
  "Cleans wafers and furnace, performs wet oxidation and measures
   oxide thickness on a test wafer in the current lot"
begin
  ...
end;
```

Figure 2-4: `Wet-oxidation` procedure outline.

The final operation shown in Figure 2-2 is a call on the procedure `pattern`, which performs a standard photolithography operation (i.e., coat, expose and develop) using the mask name supplied (in this case, NWELL). Information about the specified mask is found in the database as described in section 2.5.

BPFL is a multi-purpose language. While much of the information in a process-flow is relevant to all uses of the flow, there are cases where certain information is relevant only to a particular application and would be meaningless for others. For this reason, BPFL provides a *view* mechanism to indicate which information is appropriate to a particular set of applications. For example, information needed to simulate the process (e.g., information about specific device structure) that is not meaningful to the fabrication process is specified in a *simulation* view. An example of the use of views is shown in Figure 2-5. The **viewcase** statement specifies which operations are visible in different views. The `user-dialog` procedure called in the `fabrication` view in Figure 2-5 presents a query to an operator asking for an oxide thickness measured on the current lot of wafers. In the simulation view, the operation determines the oxide thickness from the wafer state model maintained by the simulation interpreter (see section 2.4).

```
/* In measure-oxide-thickness procedure */
viewcase
  when fabrication do
    result := user-dialog(frame: 'nanospec, operation: 'measure-oxide-thickness);
    tox := getf(result, :measured-thickness);
  end;
  when simulation do
    /* Query wafer-state model for oxide thickness */
    tox := material-attr(find-surface-segments(...),...);
  end;
end;
```

Figure 2-5: BPFL views example.

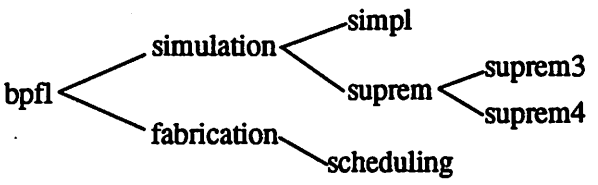

Figure 2-6: BPFL view hierarchy.

BPFL has a hierarchy of views as shown in Figure 2-6. An interpreter specifies which views of the process flow it wishes to see. Only operations in these views are visible to the interpreter. For example, an interpreter that requests a `simpl` view sees a subset of the operations in the `simulation` view. The `bpfl` view is the implied view for all code. Additional views can be defined if necessary.

The **`view`** statement is a special-case version of a `viewcase` statement. It allows the specification of code for just one view as in:

```
view fabrication do
  fabrication view operations;
end;
```

which specifies operations in the *fabrication* view. Views may also be defined in terms of logical operations on the defined view names. For example, the following statement:

```
view (simulation and not(suprem3)):
  operations for all simulation views except SUPREM3;
end;
```

specifies operations for all interpreters supporting the *simulation* view except those supporting a *suprem3* view.

Since the fabrication view is the view under which device fabrication occurs, it is the primary view of interest to the WIP system. It is discussed in greater detail in chapter 3.

BPFL process flows access standard libraries of functions using the `require` declaration. For example, the `pattern` procedure is found in the `litho` library, and a process flow uses the `litho` library by including the declaration:

```
require(litho, version: latest);
```

The `version` argument indicates which version of the library to use, and is explained in more detail in chapter 4.

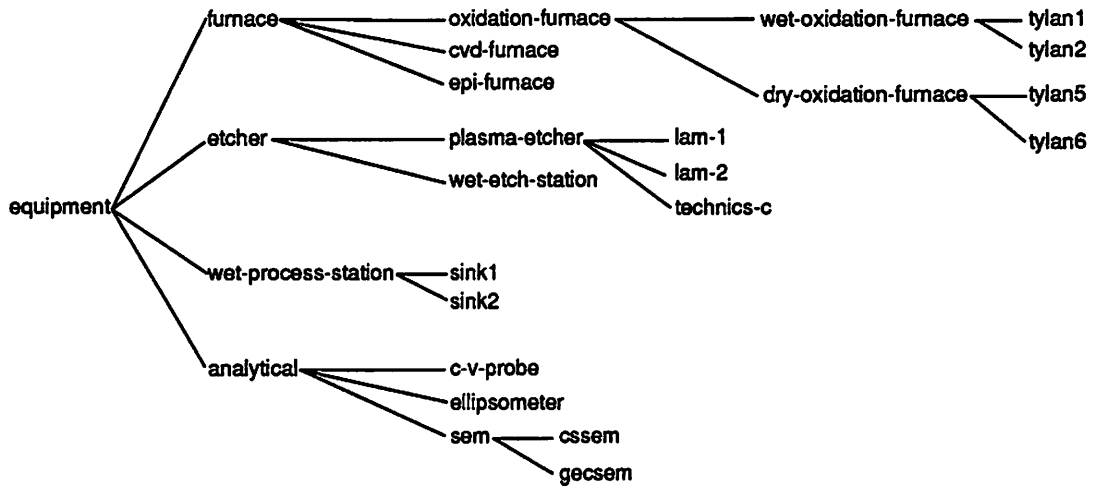

Figure 2-7: Equipment class hierarchy.

2.3 Equipment Abstractions

This section describes the BPFL statements provided to support manufacturing equipment. The equipment in a fab is described in the database so that equipment-specific operations and settings can be separated from the operations to move lots, communicate with an operator, and perform other housekeeping operations. An object-oriented design is used to facilitate the addition of new types and equipment instances to the system. A subset of the equipment class hierarchy is shown in Figure 2-7.

Equipment is allocated by using the **`with-equipment`** statement. This statement takes the name of a specific piece of equipment (e.g., `lam-1`), the name of a category of equipment (e.g., `oven`), or a list of the equipment names or categories desired and allocates a free piece of equipment that satisfies the specification. The equipment allocation procedure also implements the equipment reservation or scheduling policy. 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.

After a particular piece of equipment has been allocated, operations are performed either directly by communicating with the equipment or indirectly by communicating with an operator. The `user-dialog` procedure is used to communicate with operators and the `run-recipe` procedure is used to communicate with equipment. Figure 2-8 shows a BPFL procedure that implements a nitride etch operation. The `plasma-etch-nitride` procedure takes two arguments,

```
defflow plasma-etch-nitride(power:, overetch: = {5 %})
  "Remove nitride on wafers"
begin
  ...
  viewcase
    when simulation do
      ...
    end;
    when fabrication do
      ...
      with-equipment x of-type 'technics-plasma-etcher do
        run-recipe(x,'etch-nitride, power: etcer, overetch: overetch);
      end;
    end;
  end;
  ...
end;
```

Figure 2-8: Equipment operation example.

power and overetch. The procedure allocates a plasma etcher and executes the appropriate equipment recipe.

Equipment definitions include information about recipes. For example, the definitions for three instances of equipment type technics-plasma-etcher are shown in Figure 2-9. The first **defequipment** describes the basic properties of technics-plasma-etcher. The arguments to **defequipment** are: the name of the equipment, a list of *root-classes* from which this piece of equipment inherits properties (e.g, etcher), and equipment attributes. In this case, the only attribute is recipes which takes a list of recipes. Recipes named ash-resist and etch-nitride are defined. For example, the ash-resist recipe has a default power of 300 Watts and

```
defequipment technics-plasma-etcher ((etcher),
  recipes: (ash-resist: (gases: (#m(oxygen, flow-rate {51.1 sccm})), power: {300 W}
                         pressure: [{270 mTorr} {280 mTorr}], time: {7 min}),
            etch-nitride: (gases: (#m(helium, flow-rate: {13.0 sccm}),
                           power: {100 W}, ...))
            ...));

defequipment technics-c ((technics-plasma-etcher),
  recipes: (ash-resist: (frame: strip-resist)
            etch-nitride: (frame: etch-nitride...)
            ...);

defequipment technics-d ((technics-plasma-etcher),
  secs-device: 5,
  recipes: (ash-resist: (secs-handler: secs-technics-resist-ash),
            etch-nitride: (...)
            ...));
```

Figure 2-9: Equipment definition example.

a default time of 7 minutes. These attributes supplied in the **defequipment** are defaults which may be over-ridden by arguments to run-recipe. For example, in Figure 2-8, the values of the arguments power and overetch override values specified in the equipment definition.

If direct communication with equipment is possible, the WIP system uses Wood's SECS server [44] to download and execute recipes. For example, suppose the **with-equipment** statement in Figure 2-8 allocated technics-d. The **defequipment** that describes technics-d in Figure 2-9 shows that it is connected to the network. The secs-device attribute specifies the device address of the equipment. An additional recipe attribute secs-handler is supplied. This attribute gives the name of a procedure that is responsible for handling equipment communication for the given recipe. SECS implementations for equipment are often idiosyncratic and in general a unique handler is required for each type of equipment. In this example, run-recipe will call the procedure secs-technics-resist-ash to handle the communication.

The definition of technics-c in Figure 2-9 does not contain any information about SECS communication because direct communication with technics-c is not possible. If technics-c is allocated, run-recipe generates a user-dialog call. The name of the appropriate user-interface frame for display to the operator is specified by the frame attribute for the required recipe. More information about the user-interface to the WIP system is presented in chapter 4.

2.4 Wafer-State Representation

An SPR interpreter must maintain a description of the state of wafers. This wafer-state information is useful for three reasons. First, sanity checks may be performed on wafers before they undergo certain operations. For example, wafers with photoresist on them cannot undergo high-temperature processing so it is important to ensure that any wafers about to undergo a high-temperature step have been stripped of resist. Second, wafer-state information is useful as an adjunct to analytical measurements. An example of this use was seen in Figure 2-5, where the code to measure oxide thickness for the *simulation* view was able to query the wafer-state model. Third, wafer-state information permits wafers to be moved between process flows without loss of information about the prior treatment of wafers.

BPFL provides abstractions for material specifications, masks, and wafer profiles, in order to support the wafer-state model.

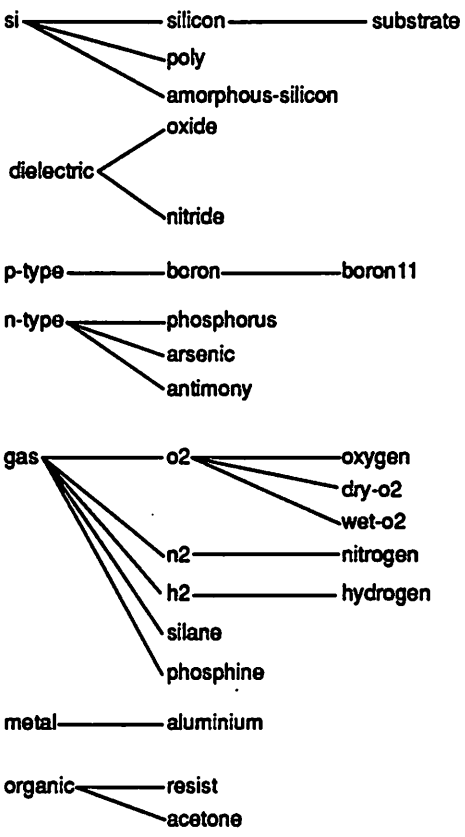

Figure 2-10: BPFL material hierarchy.

A subset of the default class hierarchy for materials is shown in Figure 2-10. Material classes have attributes that describe the properties of the material and the names that simulators use for the material. For example, the class representing polycrystalline silicon is `poly`, which is defined by:

```
defmaterial poly((si), crystal: 'poly, simpl-name "POLY",
                 suprem-name: "POLY");
```

The first argument to **`defmaterial`** is `'(si)'`, which indicates that this material class inherits properties from the `si` material class. The `crystal` argument specifies that this is a polycrystalline material. The names used for polycrystalline silicon in the SIMPL and SUPREM simulators are also given. Materials are specified in BPFL process-flows by using the `#m` shorthand. For example, the value returned by

```
#m(poly, grain-size: [(1 um), (10 um)])
```

is an object describing `poly` material with a grain size between 1–10 μm.

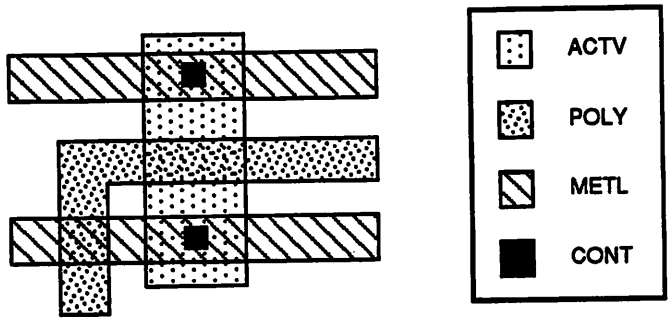

Figure 2-11: Mask layout example.

A wafer-state representation must have some knowledge of the masks used to define the wafer profile. A simple example will be discussed in order to illustrate the relationship between masks and wafer profiles. Figure 2-11 shows a simplified view of the layout of an MOS transistor. Four masking levels are shown: the active device area mask (ACTV), the polysilicon mask (POLY), the metal mask (METL), and the contact mask (CONT). Given that a single IC may have over one million transistors, it is clear that each mask is quite complex and a complete representation of mask layout would require a prohibitive amount of storage. In the BPFL representation of mask layers, two attributes are always defined: `name` and `location`. `Name` specifies the name of the mask used in the process flow (e.g., ACTV), and `location` specifies a set of design layers that are used to indicate which portions of the wafer the mask covers. `Location` values are expressed as predicates that indicate which regions of the mask are clear and which are dark. For example, the expression `NOT(CONT)` specifies that the shaded areas of the design layer CONT are inverted, which means those areas are clear on the mask. In processing terms, if CONT were the contact-definition mask, `NOT(CONT)` would be the dark-field contact-definition mask. Mask operations are used as arguments to procedures that perform photolithography operations. An example of a mask definition is:

```
defmask CONT(dark-field: t, type: chrome);
```

which specifies a dark-field mask named CONT.

The Boolean operation `OR` specifies area union and the Boolean operation `AND` specifies area intersection. For example, the expression `AND(ACTV, POLY)` describes the regions of the wafer where polysilicon crosses an active area. This expression represents the transistor gate region in a typical MOS process. BPFL can assign attributes to regions specified in terms of layout.

Function	Description
`find-segments`	Return a list of segments in the specified snapshot that have the desired properties.
`find-segments-in-lot`	Same as `find-segments` but works with snapshots in a particular lot.
`find-surface-segments`	Restrict `find-segments` to segments adjacent to ambient.
`deposit-in-lot`	A surface segment with the given attributes is added to all snapshots.
`etch-material-in-lot`	All surface segments made of the given material are removed completely where exposed.
`grow-in-lot`	A segment with the given attributes is added to all snapshots at the specified location. Differs from `deposit-in-lot` in modifying underlying silicon segments to reflect their consumption by the new segment.
`split-segments-in-lot`	Segments that overlap the given location are split in each snapshot. One new segment is within the specified location, the other is outside.

Figure 2-12: Procedure to manipulate PIF structures.

Wafer profiles are stored using a simplified version of the Profile Interchange Format (PIF) [45]. BPFL uses a subset of PIF called *naive PIF* to represent the adjacency relationships between materials on a wafer. BPFL interpreters maintain a data structure called a *snapshot* to describe the profile of a wafer. For example, the procedure `bare-silicon-wafer` used in Figure 2-2 creates a snapshot of a wafer that is composed of a single layer of material (silicon) with values given to the `crystal-face`, `resistivity`, and `dope` material attributes. Operations are provided to change snapshots to simulate the effects of processing operations (e.g., depositing material, removing material, etc.). `Grow-in-lot` and `etch-material-in-lot` are examples of high-level procedures for manipulating PIF structures. A list of some of these procedures is given in Figure 2-12. Snapshots are represented by CLOS objects that are designed to minimize storage overhead. A complete description of the data abstraction for the BPFL implementation of PIF is given in Appendix A.

A simple example of a PIF structure is shown in Figure 2-13. The left-hand part of the figure is a block diagram of the wafer profile described by the PIF snapshot. It represents a wafer (cross-hatched) covered in oxide which has been spin-coated with resist. This particular snapshot is a good representation of the state of wafers just after being coated with resist within the `pattern` operation in Figure 2-2. The `ambient` layer is a special layer representing the ambient conditions around the wafer (e.g., room temperature air, 900 °C wet-O_2, etc.). Oxide is normally grown on all

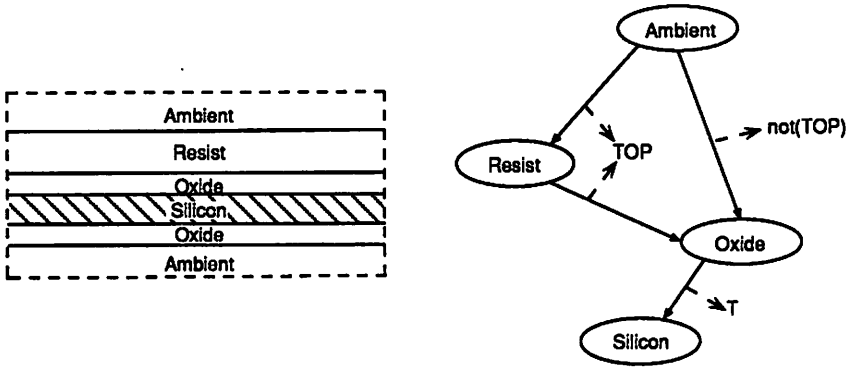

Figure 2-13: Simple wafer profile and corresponding snapshot.

surfaces of a wafer, so the back-side of the wafer is also coated with oxide, although ordinarily the back-side of the wafer is not shown in a wafer profile diagram.

The right-hand portion of Figure 2-13 shows a graphical depiction of the PIF snapshot for this wafer. PIF snapshots are composed of *segments*, *attributes*, and *boundaries*. Segments specify the information about a region or layer in a profile. They are represented by ovals. This snapshot has three segments corresponding to the layers in the material. The `ambient` segment is a system-supplied segment. For visualization purposes, the segments are shown labelled with the names of the materials making up the segment.

A *boundary* specifies that one segment is adjacent to another segment. Boundaries are represented by solid lines. The profile in Figure 2-13 has four boundary lines which represent the fact that each layer (except `silicon`) is on top of another and that the backside of the wafer is also covered with oxide and exposed to the ambient. The direction of the arrow indicates which segment lies above another segment (e.g, the arrow between `resist` and `oxide` points toward oxide because resist lies *above* oxide).

The dotted arrows on the boundaries indicate the location predicates that specify where the boundaries exist. For example, oxide bounds silicon everywhere (since oxide grows on all surfaces of the wafer), so the predicate is `T`, which represents the value *true*. Oxide is exposed to ambient everywhere except on top of the wafer (where resist lies between it and ambient), so ambient bounds oxide everywhere except the top (i.e., `not (TOP)`).

Attributes are used to specify properties about the profile (e.g., material name, whether the resist is exposed, etc.). They are represented by keyword-value pairs. Attributes may be attached to segments, boundaries, and other attributes.

Using this data structure, it is possible to query the PIF model for information about the wafer described in a snapshot. Consider once again the code in Figure 2-2 and assume that the code is being executed by a WIP interpreter for fabrication. The call on the procedure `wet-oxidation` is passed a desired oxide thickness of 100 nm. Within the procedure, a new segment for the grown oxide will be created and the default oxide thickness will be recorded as a segment attribute:

```
grow-in-lot(#m(silicon), material: #m(oxide, nominal-thickness:
                                      nominal-thickness));
```

where the argument `nominal-thickness` will have the value `{100 nm}` at execution time. `Grow-in-lot` is a procedure that simulates the effect of growing a layer of material on all wafers in the current lot. The first argument to `grow-in-lot` is the specification of the region where the growth is to occur. `#m(silicon)` is used in this case because oxide grows on silicon segments exposed to ambient containing oxygen. Oxide also grows on polysilicon about 2.5 times as fast as on monocrystalline silicon[1], so the following call to `grow-in-lot` is also required:

```
grow-in-lot(#m(poly), material: #m(oxide, nominal-thickness:
                                   2.5 * nominal-thickness));
```

Once the oxidation is complete, the oxide thickness will be measured using `measure-oxide-thickness`. As seen in Figure 2-5, the operation in the WIP view is to ask an operator to measure an oxide thickness. The value returned by the operation is stored in a local variable `tox`. This value will be assigned to the segment for the oxide:

```
seg := find-surface-segments-in-lot(material: #m(oxide));
mat := segment-attribute(seg, :material);
material-attribute(mat, :measured-thickness) := tox;
```

This information is then available for use later in the process flow. It is also available for analysis by other programs and systems.

`Find-surface-segments-in-lot` returns all segments with the specified attributes. It is possible to supply additional attributes if more selectivity is required. One common attribute used is the step-path, which is the string formed by concatenating the names of the nested steps at

[1] The differential oxide growth rate on polysilicon depends on both the dopant concentration in the polysilicon and the thickness of the oxide. The value 2.5 is often used as a "rule of thumb."

```
defflow expose-resist(mask-name:)
  "Expose wafers "
let layer := find-layer(mask-name); /* Layer corresponding to the mask */
    exposure-location := intersect-layers(top-side(),
                                          invert-layer(layer));
    old-segments := nil;
    new-segments := nil;
begin
  viewcase
    when fabrication do
      with-equipment s of-type 'stepper do
        run-recipe(s, 'expose, mask-name: mask-name);
      end;
    end;
  end;
  old-segments :=
    find-segments-in-lot(material: #m(resist, exposed: nil));
  new-segments :=
    split-segments-in-lot(old-segments, location: exposure-location);
  segment-material-attribute-in-lot(new-segments, :exposed) := t;
end;
```

Figure 2-14: `Expose-resist` procedure definition.

the current execution point. For example, the step-path for the call on `wet-oxidation` in Figure 2-2 is "`WELL-FORMATION/INIT-OX`." Any time a segment is created, the step-path is automatically added as an attribute, so one way to ensure that the desired segment is obtained is to specify the step path where it was created:

```
seg := find-surface-segments-in-lot(
                material: #m(oxide),
                step-path: "WELL-FORMATION/INIT-OX");
```

Consider the same code from Figure 2-2 running in simulation. In this case, the `measure-oxide-thickness` procedure will query the PIF data model to extract the simulated oxide thickness and assign it to the local variable `tox`, assuming the simulator has simulated the oxidation. In many situations, a time-consuming oxidation simulation is not necessary, and in these cases simulators can query the PIF data model to extract the `nominal-thickness` assigned by the `wet-oxidation` procedure. An example of where the latter might be useful is in a simulator designed to extract a profile view of part of a wafer, such as SIMPL [3].

A more complex example of PIF will now be presented. Figure 2-14 contains the `expose-resist` procedure which exposes photoresist in a masking operation. This procedure takes a `mask-name` argument which specifies the mask to use. Four local variables are used in the procedure:

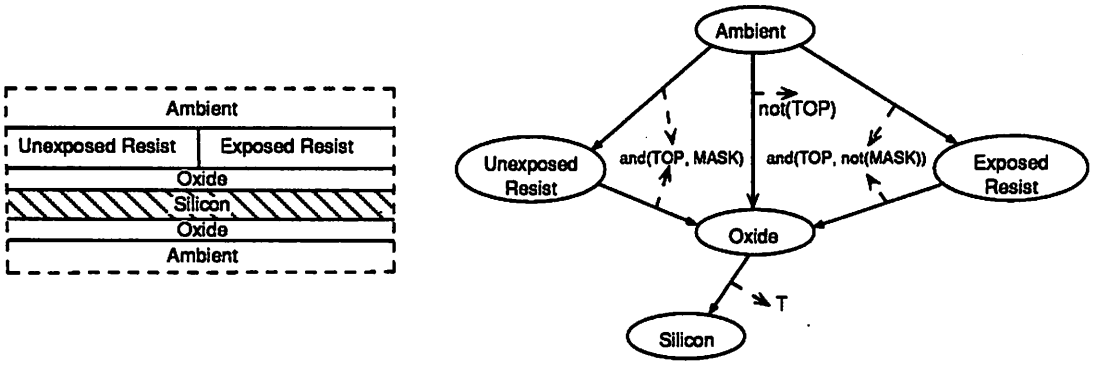

Figure 2-15: Exposed wafer block diagram and PIF snapshot.

1. `layer` - the layer describing the specified mask,
2. `exposure-location` - a specification of the clear areas on the mask,
3. `old-segments` – a variable to hold the list of PIF resist segments before the exposure, and
4. `new-segments` - a variable to hold the list of PIF segments created by the masking operation.

The clear area is calculated from the dark area specification given in the layer object. The body of the procedure executes the expose operation and modifies the PIF snapshot. Although it is not enforced by the system, it is important that the operations to change the PIF snapshot follow the operations to carry out the fabrication operation so that changes to the snapshot will not have to be undone if the fabrication operation fails.

The operations that change the PIF snapshot split the resist segment into exposed and unexposed segments. If the starting wafer has the profile shown in Figure 2-13, then the resulting profile after the `expose-resist` operation will be the one appearing in Figure 2-15. The segment with a `resist` material attribute has been split into two segments using the `split-segment-in-lot` procedure. One of the resulting segments has the `exposed` attribute of the `material` attribute set to `t` to reflect the fact that it has been exposed. The location attributes are calculated by using the mask shading attribute.

Now consider the definition of the `develop-resist` procedure in Figure 2-16 which specifies operations in two views (i.e., *simpl* and *fabrication*) to develop the resist. The *simpl* view operation generates the SIMPL simulator input for a develop operation. The *fabrication* view op-

```
defflow develop-resist()
  "Develop resist in lot"
begin
  viewcase
    when simpl do
      simpl-op("DEVL","ERST");
    end;
    when fabrication do
      with-equipment d of-type 'developer do
        run-recipe(d, 'develop-resist,
                   resist-name: material-name(resist-in-lot()));
      end;
    end;
  end;
  etch-material-in-lot(#m(resist, negative: nil, exposed: t), t);
  etch-material-in-lot(#m(resist, negative: t, exposed: nil), t);
end;
```

Figure 2-16: `Develop-resist` procedure definition.

eration executes a particular recipe on a developer. After the operations are specified, the changes to the wafer state are specified. Resist comes in two types: *negative* and *positive*. Negative resist is removed by development where it is unexposed, and positive resist is removed by development where it is exposed. In other words, a wafer coated with negative resist will be a photographic negative image of the exposed mask when developed, and a wafer coated with positive resist will be a photographic positive image of the exposed mask. In order to accommodate both resist types, two lines of code are used, one for each resist type. Figure 2-17 shows the wafer block diagram and snapshot for a developed wafer, assuming negative photoresist.

The SIMPL input-generator interpreter produces the following code when it is run on the `expose-resist` and `develop-resist` procedures:

```
EXPO mask-name no ERST

DEVL URST
```

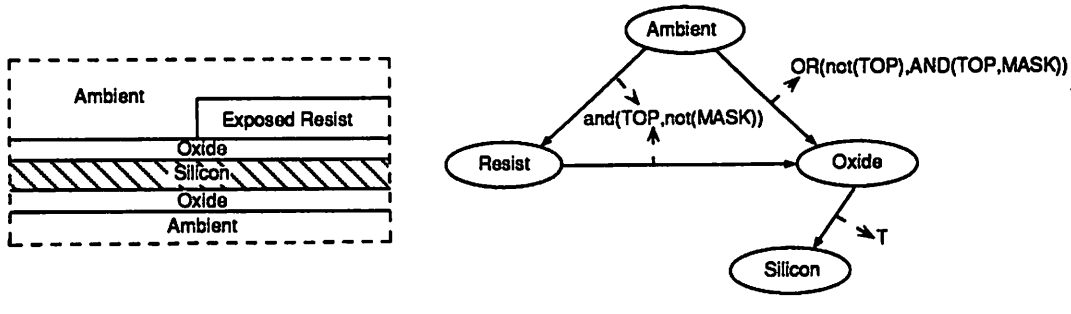

Figure 2-17: Developed wafer block diagram and PIF snapshot.

The EXPO operation is generated when the `expose-resist` procedure is interpreted and the DEVL operation is generated when the `develop-resist` procedure is interpreted. The *mask-name* in the EXPO operation 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 unexposed resist. The DEVL operation takes one argument, which specifies the resist to be removed, in this case the unexposed resist URST.

PIF snapshots are also useful for performing sanity checks on the state of wafers. Figure 2-18 shows a section of code from the `furnace-run` procedure. The `furnace-run` procedure is always called before any furnace operation occurs (i.e., all recipes defined on furnaces are dispatched using `furnace-run`). Photoresist cannot tolerate high temperatures. If a wafer is placed into a furnace with photoresist on it, the photoresist will decompose, contaminating the furnace. The purpose of the code shown in Figure 2-18 is to abort processing if resist is discovered on wafers that are about to be placed in a furnace. While this should never happen, in practice mistakes like this are fairly common, particularly in *Application-Specific Integrated Circuit* (ASIC) fabs and research fabs where processing volumes are low, several different processes are run, or processes are subject to rapid change. The ability of BPFL to perform such checks on wafer state provides an extra degree of safety, but only if used correctly.

One final use for wafer state information is to record the information about the processing history of wafers. Wafers can be processed to a particular stage using one process flow, and then transferred to a different process flow with all of the state information maintained. This activity is quite common in research fabs, although it is never done in production fabs.

The version of PIF available in BPFL is not capable of describing general device structures on a wafer because it has no notion of segment adjacency in the horizontal plane. This choice was

```
defflow furnace-run (...)
begin
  /* Can't have resist in the furnace */
  if find-segments-in-lot(material: #m(resist)) then
    raise-exception(...);
  ...
end;
```

Figure 2-18: `Furnace-run` procedure definition.

made to limit the amount of information contained in a snapshot. For processing purposes, naive PIF has been adequate for all situations observed, since processing operations never depend on queries of device structure. Physical simulators that require knowledge of two- or three-dimensional device structures (e.g., PISCES [38]) cannot be expressed completely in BPFL because additional information needs to be specified. The logical repository for this information is the CAD database used by IC designers to specify the circuit being manufactured (e.g., OCT [46]). The information in the CAD database is used to generate mask detail.

2.5 Database Entities

This section describes the database schema designed to support the statements and data types described in the previous sections.

Database entities from BPFL can be divided into two main groups. The first group contains entities required to implement a BPFL program (e.g., process flows, wafer snapshots and lots). The second group describes the fabrication facility and contains entities that a BPFL program might access (e.g., equipment and mask descriptions).

The database schema for run state is shown in Figure 2-19. The `run` class describes the basic information about a run: its name, the process flow used by the run, the current run step-path and the run *stack*. The stack consists of a description of the process-flow execution state maintained by the WIP system interpreter discussed in chapter 5. The `process-flow` class contains the name and version of a process flow, as well as a brief description of the flow and maintenance information about it such as the last modification time, the user responsible for maintaining the process-flow, and a list of users authorized to use the process flow.

```
run(name: string, id: integer, process-flow: process-flow*, step-path: string,
    stack: stack*);

process-flow(name: string, version: string, id: integer, modification-date: datetime,
             maintainer: user*, description: string, authorized-users: user*);
stack(root-frame: frame*, ...);

wafer(run: run*, id: integer, index: integer, scribe: string, snapshot: snapshot*);
lot(run: run*, id: integer, name: string, wafers: wafer*[]);

snapshot(run: run*, id: integer, name: string, segments: segment*[]);
segment(id: integer, boundaries: boundaries*[], attrs: attrs*[]);
boundary(id: integer, attrs: attrs*[]);
attr(id: integer, name: string, value: string, attrs: attrs*[]);
```

Figure 2-19: Database schema for run state.

```
equipment(name: string, parent: equipment[]*, secs-address: integer,
          recipes: recipe*[]);
recipe(name: string, attributes: string);

mask-set(name: string, masks: mask*[]);
mask(name: string, type: mask-type, dark-field: logical);
```

Figure 2-20: Database schema for facility description.

The `wafer` class includes the wafer name and a pointer to the snapshot description of the wafer. The `lot` class has pointers to all wafers that belong to that lot. The wafer state information is stored in four classes called `snapshot`, `segment`, `boundary` and `attr`, which describe the PIF entities making up snapshots.

The database description for some classes that describe the facility are shown in Figure 2-20. The `equipment` class includes the equipment name, the `parent` classes of equipment from which this instance inherits, the `secs-address` for the equipment, and pointers to equipment recipes. Recipes are stored in a `recipe` class with a `name` and a property list of attribute-value pairs.

Mask-sets are stored in a `mask-set` class, with a name and pointers to the masks belonging to the set. The `mask` class includes the mask name, the mask type (i.e., chrome or emulsion) and a logical value indicating whether or not the mask is dark field. This value is used to determine the `location` of the mask.

2.6 Summary

The basic goals of BPFL are to provide a complete, facility-independent process-flow representation. BPFL is a Lisp-based procedural language with support for common abstractions encountered in processing, such as wafers, lots and equipment. BPFL uses views to provide domain-specific information where necessary. A version of PIF is used to maintain wafer state information. BPFL stores all information about process flows and facilities in a database.

Chapter 3
BPFL Statements for Fabrication

This chapter describes BPFL statements to support IC fabrication. Fabrication places special demands on an SPR. First, some means of communicating with operators and equipment is required. Second, events that occur and data collected during processing must be recorded to maintain run history. Third, errors and unexpected events often occur in processing so an exception-handling mechanism is required to deal with them. Fourth, rework loops are common in real-world processes so it is worthwhile to provide a language statement to express them explicitly. Finally, constraints may be placed on operations (e.g., time limits between operations such as the constraint that a nitride deposition in a LOCOS[1] step should be started within 30 minutes of completing the preceding oxidation step).

The statements described in this chapter are normally only used by the WIP interpreter, although timing constraints have obvious applications in scheduling.

3.1 Equipment Communication

BPFL equipment abstractions have already been introduced in chapter 2. The fundamental equipment allocation and access statement is **`with-equipment`**. For example, the following fragment of code:

```
with-equipment e of-type equipment-specification do
    body;
end;
```

allocates a piece of equipment that satisfies the *`equipment-specification`*, assigns that instance of the equipment to the variable `e`, executes the operations in the *`body`* and then deallocates the equipment. It is important that allocated equipment always be deallocated. The **`with-equipment`** statement guarantees that equipment will be deallocated, even if errors occur while processing the code in `body`. This is achieved through the use of the Lisp primitive `unwind-protect` which guarantees the execution of cleanup operations after a code body even if the body generates an exception. Figure 3-1 shows the semantics of the **`with-equipment`** statement. The code spec-

[1] LOCal Oxidation Of Silicon is an isolation technique.

```
unwind-protect
  e := allocate(equipment-specification);
  body;
  cleanup deallocate(e);
end;
```

Figure 3-1: With-equipment semantics.

ified in the **cleanup** clause is guaranteed to execute.

Once equipment has been allocated, it may be accessed using the Semiconductor Equipment Manufacturers Institute (SEMI) Equipment Communications Standard protocol (SECS). The BPFL WIP system uses a SECS server developed by Wood [44]. The interface to the server is via a Common Lisp package [47]. For example, the following BPFL code opens a connection with a particular tube in a Tylan furnace bank:

```
m := create-message(DMSTREAM, DMCONNECT, needs-reply: t,
                    device: tylan-address,
                    body: create-item(ASCII, "furnace 2"));
write-message(m);
```

`Create-message` is a function provided by the SECS server. The first two arguments to `create-message` specify the `STREAM` and `FUNCTION` codes for the message. The stream code identifies a particular class of messages with similar purposes and the function code identifies a particular message type within that class of messages. The `needs-reply` argument is used to indicate whether the SECS server should wait for a reply from the recipient of the message. `Device` specifies the SECS device address of the message recipient, in this case the tylan furnace controller. `Body` specifies the contents of the SECS message to send. In this example, the `body` is a SECS `item` consisting of the string of characters `"furnace 2"`.

The `Write-message` function sends the message to the recipient. In this example, the message is sent to the tylan furnace controller and connects the server to the second tube within the furnace bank attached to the controller.

For higher-level equipment access, the `run-recipe` procedure provided by the WIP interpreter is normally used. This procedure uses the information specified in equipment definitions stored in the database (see Figure 2-9) to execute high-level procedures to communicate with equipment. The `secs-device` and `secs-handler` equipment and recipe attributes are used for this purpose as described in section 2.3.

Every equipment operation is logged. Log records include the start and end times of the operation. The procedure `last-equip-time` returns the completion time of the last operation carried out on a lot of wafers by a given type of equipment or recipe. For example, the procedure call:

```
last-equip-time(lot: 'product, equipment: 'hmds-tank, recipe: 'hmds-coat);
```

returns the completion time of the last hmds-coat operation performed on wafers in the `product` lot. When called without arguments, `last-equip-time` returns the completion time of the last equipment operation on the current lot of wafers.

3.2 Operator Communication

The `user-dialog` procedure can be used to communicate with an operator. In the prototype WIP system, this procedure calls an ABF frame. An ABF frame contains two elements: first, a *form* through which data can be displayed to or entered by the operator; second, a *menu* of operations that he or she can execute [48]. The form describes the operation to be performed and entry fields in which the operator can enter measurements and status data. The operations provided allow the operator to abort an operation, signal completion of an operation, check equipment status, and disconnect from a run.

As an example of `user-dialog`, consider the `inspect-resist` procedure used in photolithography. The operator enters wafer identifiers and inspection results into the form in the `Inspect-Resist` frame shown in Figure 3-2. This frame is called by the `user-dialog` procedure. The data entered by the operator is returned to the procedure as a *property list* of attribute-value pairs. The `getf` procedure is used to extract attributes from the property list. In this case, `user-dialog` returns a value with `rework` and `scrap` attributes, which specify which wafers should be reworked and scrapped, respectively.

Figure 3-3 shows an implementation of the `inspect-resist` procedure used in photolithography. The procedure allocates a microscope and asks the operator to inspect each wafer in the lot. The `user-dialog` function call results in the display of the `inspect-resist` frame. The value returned by `user-dialog` is queried using `getf` to extract the `rework` and `scrap` attributes and force rework to occur if necessary.

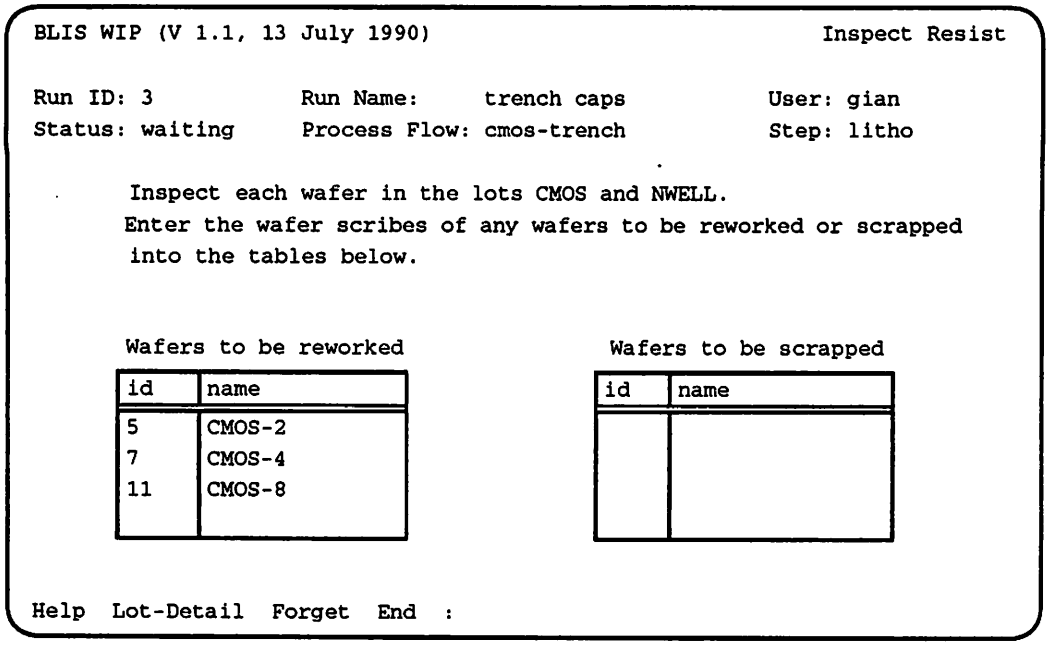

Figure 3-2: `Inspect-Resist` frame.

3.3 The 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 CLOS objects of different classes. Each class represents a different type of event that is being logged. A log record is called a log object (LO). Figure 3-4 shows the class hierarchy. Figure 3-5 shows the database conceptual schema for some different types of log objects. New types of LOs can be defined by creating a class for the LO types.

```
defflow inspect-resist()
  "Inspect each wafer and put wafers to be reworked into the rework lot
   and wafers to be scrapped into the scrap lot."
begin
  view fabrication do
    with-equipment scope of-type 'microscope do
      let results := user-dialog(name: 'inspect-resist,
                                 equipment: scope)
      begin
        wip-log('Resist-Inspect, results);
        move-sublot(getf(results, rework:), 'current, 'rework);
        move-sublot(getf(results, scrap:), 'current, 'scrap);
        if lot('rework) then raise-exception('rework);
      end;
    end;
  end;
end;
```

Figure 3-3: `User-dialog` procedure example.

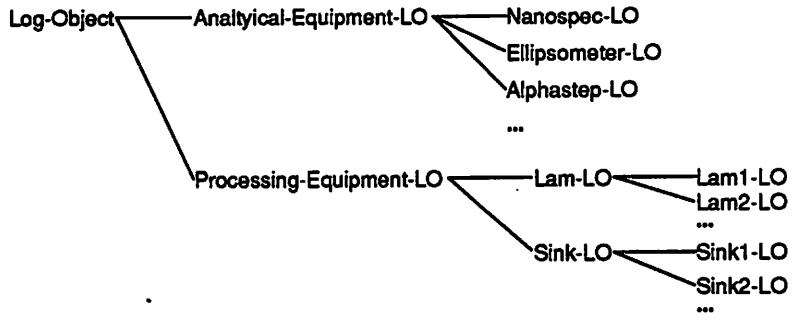

Figure 3-4: WIP log object class hierarchy.

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. For example, the `inspect-resist` procedure mentioned in the previous section writes a `Resist-Inspect-LO` to the log that records the status entered by the operator for each wafer inspected.

A WIP LO 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 `time` attribute specifies the date and time when the object was written, and the `tag` attribute records a value included in the operation that writes the log object. The `tag` attribute is

```
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);
Analytical Equipment Log Objects
  Analytical-Equipment-LO() inherits (Log-Object);
  Nanospec-LO(thickness-array: unit[]) inherits (Analytical-Equipment-LO);
  Ellipsometer-LO(thickness-array: unit[]) inherits (Analytical-Equipment-LO);
  Alphastep-LO(height-array: unit[]) inherits (Analytical-Equipment-LO);
    ...
Processing Equipment Log Objects
  Processing-Equipment-LO() inherits (Log-Object);
  Lam-LO(wafer-etch-time: datetime, recipe: string, power: unit)
        inherits (Processing-Equipment-LO);
  Lam1-LO() inherits (Processing-Equipment-LO);
  Lam2-LO() inherits (Processing-Equipment-LO);
  Sink-LO(wash-resistivity: unit, piranha-etch-time: unit)
        inherits (Processing-Equipment-LO);
  Sink6-LO(bhf-etch-time: unit) inherits (Processing-Equipment-LO);
  Sink8-LO(poly-etch-time: unit) inherits (Processing-Equipment-LO);
    ...
```

Figure 3-5: Database schema for WIP log objects.

```
/* 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 1991"
  and class(log-object) = "Ellipsometer-LO"
group by run-id
```

Figure 3-6: Sample WIP log queries.

optional. It can be used to specify a unique string that identifies the log object written by a particular operation. The string can be used to simplify the predicate required to search the log for all objects written by the operation.

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

```
wip-log('Resist-Inspect, results);
```

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 a process was run. Queries can be executed from an ad hoc query interface, an engineer's notebook interface [49], or a BPFL program. The engineer's notebook interface 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 its past history without having to create special data structures 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 3-6 shows three sample log queries specified in an extended version of SQL. 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 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.

3.4 Exceptions

Errors and unexpected events occur frequently during semiconductor fabrication. For example, a furnace may detect an abnormal gas flow during an oxidation operation and abort the operation. An SPR that is unable to cope with such abnormal events is unsuitable for use in a fabrication environment. This section describes BPFL exception-handling mechanisms.

BPFL uses the proposed ANSI standard Common Lisp conditions package [39] to define exception handlers and raise exceptions. Many exceptions are caught by the WIP interpreter itself (e.g., exceptions are used in the implementation of constraints which are discussed below). However, a user may write BPFL code to handle and raise exceptions explicitly. An exception handler is the routine that is called when a particular error occurs. An error is signalled by raising an exception which suspends execution of the BPFL program and calls the appropriate handler. The handler can:

1. change environment,
2. record events,
3. change program execution, or
4. suspend or abort a run.

Note that suspending a run can send a message to the equipment operator, place the run in a queue that is managed by a process or equipment engineer, or send an email message to the person who started the run. The action taken is determined by the policy established by a particular fab.

The language statement used to define an exception handler is `handler-case`. It is used to trap specific exceptions that occur inside a body of code. The syntax of `handler-case` is:

```
handler-case
  body;
  on-exception var-1 := exception1 do
    exception1-handler;
  end;
  on-exception var-2 := exception2 do
    exception2-handler;
  end;
  ...
end;
```

The semantics of this statement are as follows. The *body* is executed, and if one of the specified exception types (e.g., *exception1*, *exception2*,...) occurs, the specified *exception-handler* is executed. An example of the use of **handler-case** appears in Figure 3-7 which shows the `run-recipe` procedure that downloads and executes a recipe in a piece of equipment. The **handler-case** statement deals with equipment errors. The *body* of the `handler-case` is the code:

```
download-recipe(...);
start-recipe(...);
```

which actually runs the recipe. If an `equipment-error` exception is raised during the execution of this code, the handler writes an entry to the log that describes the nature of the error and processing is suspended until an operator corrects the problem. The variable `c` contains a structure, called a *condition*, that describes the error.

The operations available to change control-flow in an exception handler are:

1. `Halt-run` – Displays a user-dialog asking the user to choose the action,
2. `Resignal` – passes the exception up to the next higher-level exception handler,
3. `Restart-body` – Execute the code in the body of the **handler-case** again, and

```
defflow run-recipe(...)
begin
  ...
  handler-case
    download-recipe(...);
    start-recipe(...);

    on-exception c := equipment-error do
      report-error("Error occurred during run-recipe: ~s", c);
      halt-run();
    end;
  end;
end;
```

Figure 3-7: Handler-case example.

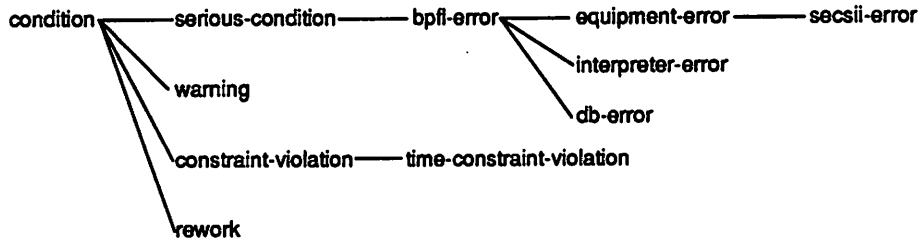

Figure 3-8: BPFL condition types.

4. `Ignore-exception` – Continue executing the code as if the exception had not occurred.

If the body of a **`handler-case`** or a procedure called in the body contains an exception-handler for a condition, the lower-level exception handler will be executed. For example, if `start-recipe` contains a **`handler-case`** for the same condition, an `equipment-error` exception signalled within `start-recipe` will over-ride the handler in Figure 3-7.

The data types that specify a condition form a hierarchy. The most general type of condition is `condition`, which has subtypes, `serious-condition`, `warning`, `constraint-violation`, and `rework`. The WIP interpreter handles all condition types defined in the Common Lisp standard [39]. Additional error types are defined by the WIP system, and they are shown in Figure 3-8.

Exceptions are signalled by using the `raise-exception` procedure. For example, an `equipment-error` exception may be signalled with the following code:

```
raise-exception('equipment-error, machine: 'tylan5,
                cause: "calibration failure");
```

An exception can be raised explicitly by a BPFL program or in response to an error returned by an equipment or user-dialog operation.

BPFL process flows can define their own types of conditions using **`defcondition`**. Figure 3-9 shows an example of defining a `tylan-error` condition for use with tylan furnace tubes. The condition has two slots: `recipe` and `step-number`. Since this condition is defined as

```
defcondition tylan-error ((equipment-error),
  "Tylan-specific error, reports furnace details"
  recipe;
  step-number);
```

Figure 3-9: `Defcondition` example.

a subtype of `equipment-error`, it inherits slots from `equipment-error`. Using `tylan-error`, an exception with more detail about the cause of the error can be generated:

```
raise-exception('tylan-error, machine: 'tylan5,
                cause: 'calibration-failure, recipe: "SWETOXB",
                step-number: 5);
```

3.5 Rework

Rework is a common operation in semiconductor processing. A rework loop in BPFL specifies the processing to be done, a test for correctness (e.g, inspecting wafers for good pattern definition in photolithography), and operations to execute on a wafer that fails the test. The **`rework-loop`** statement takes the following arguments:

1. the operations to perform (i.e., the rework body),
2. an operation to test whether the rework body was completed correctly (**`rework-test`**),
3. the number of times to retry the loop before giving up (**`retry-count`**),
4. operations to perform before retrying the operations in the body (**`rework-prefix`**), and
5. a procedure to call if the retry count is exceeded (**`retry-failure`**).

The semantics of the **`rework-loop`** statement are shown in Figure 3-10. `Body` specifies the operations to be performed. The **`rework-test`** operation tests the wafers and puts the ones that require rework into the `rework` lot and the ones that cannot be reworked into the `scrap` lot. It also returns a value indicating if all wafers passed the test. If some wafers failed the test, the wafers in the `scrap` lot are removed from the wafers allocated to the run. The loop is exited if the `rework` lot is empty. The **`retry-count`** argument is an integer that is decremented each time the rework loop is executed. If the **`retry-count`** is decremented to zero, the **`retry-failure`** operation is executed. Otherwise, the **`rework-prefix`** is executed and the wafers in the `rework` lot are processed again.

The photolithography flow shown in Figure 3-11 includes an example of rework. The basic sequence of operations is the following:

1. Dehydrate – dry the wafers to promote resist adhesion. Dehydration is usually carried out at 120 °C for 20 minutes.

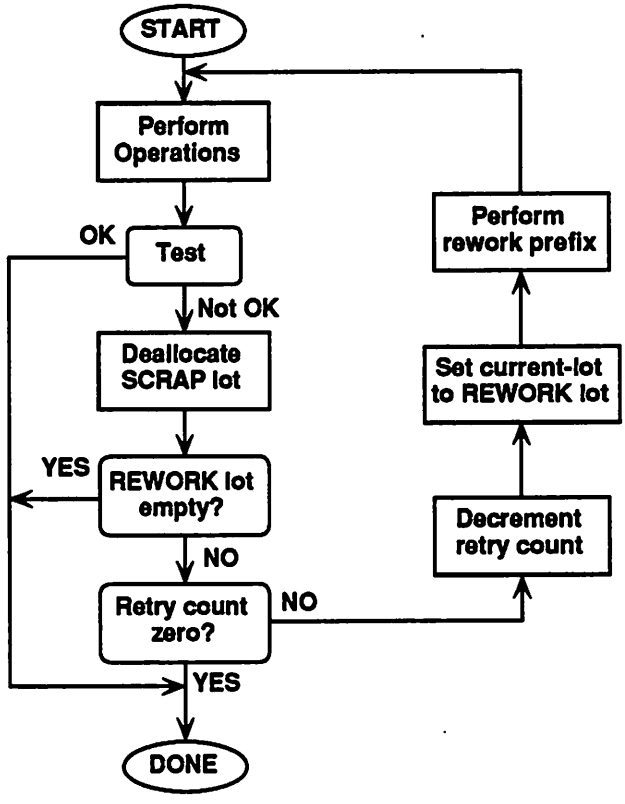

Figure 3-10: Rework semantics.

2. Apply HMDS (Hexamethyldisilazane) – this improves adhesion of resist to oxides. Wafers are placed in an HMDS ambient for three minutes.
3. Apply resist – photoresist is deposited on the wafers, which are then spun at high speed to form a uniform layer of resist. Next, the wafers are baked at 120 °C[1] for one minute to increase the viscosity of the resist layer. Finally, the resist layer is inspected for uniformity.
4. Expose – the wafers are aligned to a mask and the mask is photographed onto the resist layer.
5. Develop – the resist on the wafers is developed and the wafers are washed.
6. Inspect – The wafers are inspected to ensure acceptable mask-pattern transfer. Unsatisfactory wafers are stripped of resist and reworked.
7. Descum – wafers are etched in a low power oxygen plasma to clean up resist

[1] Exact temperatures and times depend on the photoresist used. The figures here are for Kodak-820 positive resist.

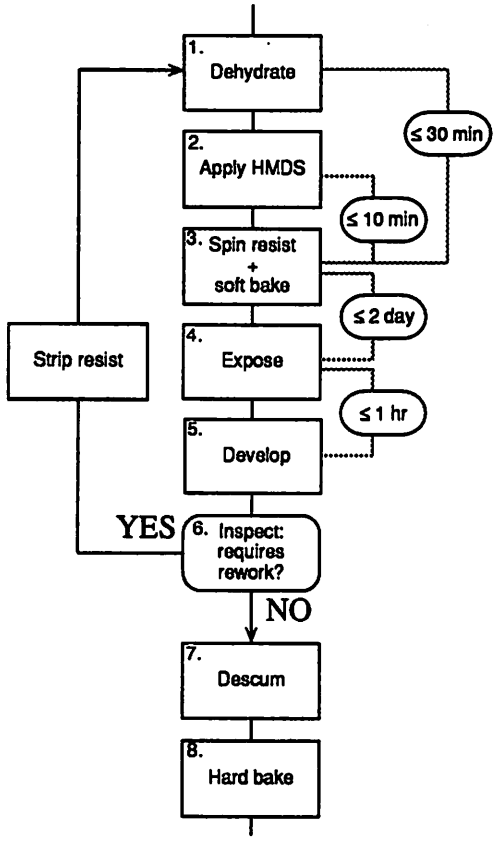

Figure 3-11: Photolithography rework loop and timing constraints.

deposits left behind on developed areas.

8. Hard bake – wafers are baked at 120 °C for thirty minutes (possibly at a higher temperature for thick resists or resists with high water content) to harden the resist in preparation for the etching step to follow.

This sequence of operations is known as *patterning*. A procedure `pattern` that implements the rework loop for photolithography is shown in Figure 3-12. The figure also includes timing constraints which will be discussed in a later section. The body of the **`rework-loop`** statement is the code:

```
spin-soft-bake(resist: resist);
expose-resist(mask-name: mask-name);
develop-resist();
```

The **`rework-test`** is the procedure `inspect-resist`, which puts unacceptable wafers into the `rework` and `scrap` lots. This example introduces an additional complication not considered before. In *double-photo* lithography, wafers are patterned even though they are already coated with resist. For example, double-photo operations are used whenever possible for high-energy or high-

```
defflow pattern(mask-name:, will-double:, resist: = *default-resist*)
 "Basic photolithography - coat, expose, develop, descum, bake"
let double-photo := find-surface-segments-in-lot(material: #m(resist));
begin
  step PATTERN do
    rework-loop
      spin-soft-bake(resist: resist);
      expose-resist(mask-name: mask-name);
      develop-resist();
      rework-test inspect-resist();
      retries 5;
      rework-prefix if not(double-photo) then
                      strip-resist();
                    end;
    end; /* rework */
    descum-resist();
    hard-bake-resist(double-photo: (double-photo or will-double));
  end; /* step */
end;
```

Figure 3-12: Pattern procedure definition.

dose implants to provide maximum resist thickness for unimplanted regions of the wafer to limit substrate damage. If double-photo is required, it is usually inadvisable to strip the resist unless gross misalignment of the mask is apparent. For this reason, the **rework-prefix** in this example is:

```
rework-prefix if not(double-photo) then strip-resist();
```

which prevents resist from being stripped if a double-photo operation is in progress. The pattern procedure queries the wafer-state model to determine if double-photo is being performed:

```
let double-photo := find-segments-in-lot(material: #m(resist));
```

This line of code assigns a non-nil value to the double-photo variable if there is any resist present on the wafers before the pattern operation begins.

Occasionally it is necessary to force rework to occur from within the code in the body of a **rework-loop**. For example, if wafers are found to be coated with an uneven layer of resist, there is no point in exposing and developing them because they will fail the inspect-resist test. Rework may be started at any time by raising the rework exception which forces the rework:

```
raise-exception('rework);
```

The rework and scrap lots must be set up correctly before the exception is raised. Rework can also be started by an operator if a run is operating within a **rework-loop**. This feature is useful for dealing with error conditions that are not handled directly by the process-flow code.

3.6 Constraints

Process flows often specify constraints on operations or between operations. Furthermore, there are general policy constraints that are imposed on certain types of processes. These constraints often cannot be checked at any one place in the process flow, because they must be true over a section of the flow. In BPFL, constraints are specified as a dynamic scope over which certain predicates must be true. If a constraint is violated, the interpreter raises an exception that will be caught by a handler that knows how to deal with the situation.

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 22 °C:

```
constrain
  body;
  when current-temperature() > (22 degC) do
    halt-run();
  end;
end;
```

The **when** clause specifies a constraint predicate which is followed by operations that are executed if the constraint is violated. Any number of **when** clauses can be specified in one **constrain** statement.

A more complicated example is shown in the photolithography flow in Figure 3-11. The constraints specified in ovals on the right side of the figure are "rule-of-thumb" timing constraints used in the Berkeley microlab [14]. The constraints applying to photolithography are the following:

1. After wafers are dehydrated, moisture in the atmosphere will quickly adsorb to their surface so resist should be applied within thirty minutes.
2. HMDS rapidly evaporates from the surface of wafers so resist should be deposited within ten minutes of HMDS application.
3. Wafers should be exposed within two days of coating to prevent problems with resist adhesion and development caused by water adsorption.
4. Wafers should be developed within one hour of exposure to prevent softening of the pattern edge due to molecular diffusion across boundaries between exposed and unexposed areas of resist.
5. The hard bake should be completed no more than an hour before the next step

is started. This timing constraint is not indicated in Figure 3-11 because the exact time limit on the constraint depends on the subsequent step, and must be specified outside the scope of the pattern procedure.

The remainder of this section describes how these constraints are specified in BPFL.

The dehydrate-wafers procedure is shown in outline form in Figure 3-13. It is called by the spin-soft-bake procedure called in pattern. The **if** statement:

```
if (min(segment-material-attribute-in-lot(segments, :dehydration-time)) + (30 min)
  < current-time()) then
```

checks the wafer-state information to see if the wafers require dehydration. The segments variable contains a list of all substrate segments of the wafers in the current lot. Substrate segments are used for the purpose of recording attributes that apply to the whole wafer, such as dehydration-time and cleanliness. Since segments contains a list, the min procedure is used to select the earliest dehydration time of the wafers in the lot. If the earliest dehydration-time is more than thirty minutes in the past, all wafers are dehydrated.[1] The wafer-state model is updated by the statement:

```
segment-material-attribute-in-lot(segments,:dehydration-time) :=
     last-equip-time();
```

which sets the dehydration-time to be the completion time of the last equipment operation. The final statement:

```
min(segment-material-attribute-in-lot(segments,:dehydration-time));
```

selects the earliest value from the list of dehydration times returned by dehydrate-wafers. This value is important for the operation of the spin-soft-bake procedure to be discussed shortly.

```
defflow dehydrate-wafers()
  " Dehydrates wafers if necessary and returns the dehydration time "
let segments := find-segments-in-lot(material: #m(substrate));
begin
  if (min(segment-material-attribute-in-lot(segments, :dehydration-time)) + (30 min)
    < current-time()) then
    ... /* Equipment operations to dehydrate wafers */
    segment-material-attribute-in-lot(segments,:dehydration-time) :=
         last-equip-time();
  end; /* if */
  min(segment-material-attribute-in-lot(segments,:dehydration-time)); /* return val */
end;
```

Figure 3-13: Dehydrate-wafers implementation.

[1] An obvious enhancement is to only dehydrate those wafers requiring it.

The pattern procedure in Figure 3-14 is the same as the code shown in Figure 3-12 except that it implements the timing constraints between the spin-resist, expose, and develop operations using the expression:

```
when (max-time-between('spin-soft-bake, 'expose-resist,
                {2 day}) or
                max-time-between('expose-resist, 'develop-resist,
                {1 hour})) do
  halt-run("time-constraint-violation in pattern");
end;
```

The max-time-between procedure takes two procedure names and a time interval as arguments.[1] The two procedures must be called within the body of the **constrain** in which the **when** clause appears. The constraint implied in the max-time-between procedure is that the time between calling the first procedure and calling the second procedure must not exceed the last argument. If it is greater, the constraint is violated and the halt-run operation is executed. The body of the **constrain** is composed of the operations to execute under the specified constraints.

```
defflow pattern(mask-name:, will-double:, resist: = *default-resist*)
 "Basic photolithography - coat, expose, develop, descum, bake"
let double-photo := find-surface-segments-in-lot(material: #m(resist));
begin
  step PATTERN do
    rework-loop
      constrain
        spin-soft-bake(resist: resist);
        expose-resist(mask-name: mask-name);
        develop-resist();
        when (max-time-between('spin-soft-bake, 'expose-resist,
                    {2 day}) or
                    max-time-between('expose-resist, 'develop-resist,
                    {1 hour})) do
          halt-run("time-constraint-violation in pattern");
        end;
      end; /* constrain */
      rework-test inspect-resist();
      retries 5;
      rework-prefix if not(double-photo) then
                      strip-resist();
                    end;
    end; /* rework */
    descum-resist();
    hard-bake-resist(double-photo: (double-photo or will-double));
  end; /* step */
end;
```

Figure 3-14: Pattern procedure implementation with constraints.

[1] Max-time-between may take a procedure name, an equipment operation, an absolute time, or a time interval as arguments.

```
defflow spin-soft-bake(resist: = *default-resist*)
 "dehydrate, hmds treat and spin resist onto wafers"
let last-dehyd-time := dehydrate-wafers();
begin
  constrain
    deposit-hmds();
    deposit-resist(resist: resist);
    when max-time-between(last-dehyd-time,'deposit-resist,
                          (30 min)) do
      last-dehyd-time := dehydrate-wafers();
      restart-body();
    end;
    when max-time-between('deposit-hmds,'deposit-resist,
                          (10 min)) do
      restart-body();
    end;
  end; /* constrain */
end;
```

Figure 3-15: `Spin-soft-bake` implementation.

The `spin-soft-bake` procedure in Figure 3-15 shows how the constraints between steps 1–3 and 2–3 in the rework loop in Figure 3-11 are specified. The procedure begins by calling `dehydrate-wafers`. As has been described, `dehydrate-wafers` does not dehydrate wafers unless necessary, but always returns the earliest dehydration time of wafers in the current lot. The value returned by `dehydrate-wafers` is used in a constraint in `spin-soft-bake`:

```
when max-time-between(last-dehyd-time,'deposit-resist,
                      (30 min)) do
  last-dehyd-time := dehydrate-wafers();
  restart-body();
end;
```

The `last-dehyd-time` variable contains the value returned by dehydrate-wafers. This constraint forces dehydration if the wafers are not coated within 30 minutes of the last dehydration time.

The constraint between steps 2-3 in the rework loop is implemented by the second **when** clause:

```
when max-time-between('deposit-hmds,'deposit-resist,
                      (10 min)) do
  restart-body();
end;
```

The exception handler forces the code in the body of the **constrain** to be executed again.

The implementation of constraints is discussed in chapter 5.

3.7 Summary

Adequate support for fabrication is an important requirement of an SPR. This chapter has presented the BPFL statements intended for fabrication. These statements include equipment and operator communication, exception handling, rework, and timing constraints.

Chapter 4
The WIP Run-Management System

This chapter describes the BPFL WIP system and the run-management system. The chapter is organized as follows. First, the architecture of the WIP system is reviewed. Then an example of using the WIP system to start and monitor a run is presented. Third, the interface to the WIP log is described. Fourth, support for modifying runs in progress is discussed. Finally, the WIP version control system is described.

4.1 WIP System Architecture

The software architecture of the WIP system is shown in Figure 4-1. The system is composed of many processes that communicate with users, equipment, and the CIM database. The main process is the WIP interpreter that executes runs. A run corresponds to an execution of a BPFL process flow. Each run is represented by data structures that contain the run state (e.g., the next statement to execute, the names and values of local variables created by the program, and data retrieved from the database). The WIP interpreter executes many runs at the same time. In other words, it is a server process.

The user interface process(es) support communication with operators. Operators at different locations in the fab can communicate with any run by connecting to the WIP interpreter through

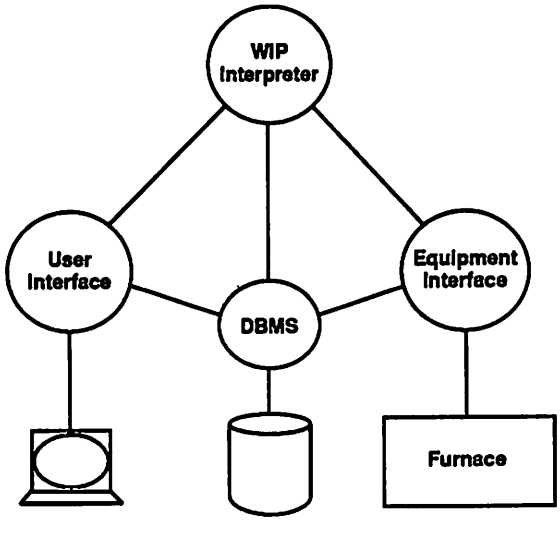

Figure 4-1: WIP system architecture.

a user interface (UI) process. BPFL `user-dialog` commands are sent to the appropriate UI process.[1] The UI process is an Application-By-Forms (ABF) [50] program in the current prototype. Every user has a separate UI process, and the WIP system prevents more than one user from connecting to any run at the same time.

The UI process uses a terminal-based interface rather than a graphical user-interface (GUI) because the Berkeley Microlab is equipped with ASCII terminals. The current terminal-based interface will be replaced with a GUI after suitable terminals are installed. Some operations that are cumbersome to perform with the current implementation (e.g., moving wafers between runs) are much easier to perform in a GUI.

The equipment interface (EI) process(es) support communication with equipment. Each EI process is an instance of Wood's SECS server [44]. An object-oriented SECS interface is defined within BPFL, and methods are defined for high-level equipment operations (e.g., run recipe, monitor run, fetch equipment status, etc.). These methods are implemented by remote procedure calls that invoke SECS commands implemented in the EI process.

All processes in the WIP system communicate with the CIM database. The WIP interpreter checkpoints the states of runs in the database so that other users and programs can access run information and active runs can be recovered if a computer or network fails. The UI process uses form definitions stored in the database and allows the user to browse the CIM database (e.g., active runs, WIP logs, etc.). The EI process accesses equipment information stored in the database.

ABF applications use *frames* as the user-interface. A frame consists of two components: a *form* that displays information to the user and in which the user enters information, and a *menu* listing the available operations that the user can execute [48]. The main frame of the UI process is the `Run-Summary` frame shown in Figure 4-2.[2] The top line of all frames in the UI process displays system information: the system version and the name of the current frame. Most of the screen area is taken up by a `Run-Information` table. This table displays a list of runs and information about

---

[1] In a low volume fab such as the Berkeley Microlab, a user moves to a different terminal and reconnects to the run. In a high volume fab, the WIP system sends the command to the user interface process at the appropriate workcell.

[2] The names of frames, operations and fields are shown in the text in monospace font with spaces replaced by hyphens.

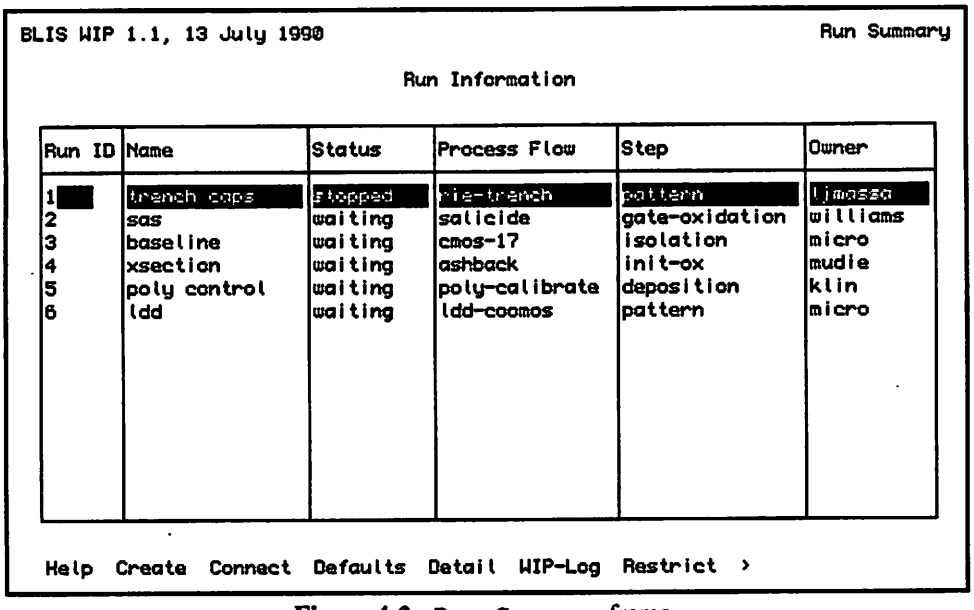

Figure 4-2: `Run-Summary` frame.

them including their status (i.e., `running`, `waiting`, `stopped`, `aborted`, `finished`), the process flow, the current step, and the run owner.

The bottom line in a frame lists the operation menu. The operations in the `Run-Summary` frame are listed in Table 4-1[1]. Every frame in the UI process has a `Help` operation. The `Help` operation gives information about keyboard mapping and a description of the frame screen layout and operations. The top portion of the descriptive text about the `Run-Summary` frame displayed by the `Help` operation is shown in Figure 4-3. The `Create` operation is used to start a run. `Connect` displays the current `user-dialog` operation for a run. Both operations are described in the next sec-

Operation	Description
`Help`	Displays help screen for the frame.
`Create`	Create a new run.
`Connect`	Connect to an existing run.
`Defaults`	Set up user defaults for the WIP system.
`Detail`	Provide more information about a run.
`WIP-Log`	Display the WIP-Log for a run.
`Restrict`	Enter criteria for runs to display (e.g., only runs owned by a particular user).
`Version`	Displays process-flow version information.
`Quit`	Leave the WIP system.

Table 4-1: `Run-Summary` frame operations.

[1] The menu for the `Run-Summary` frame is too long to fit across the screen, and the `Version` and `Quit` operations do not appear in the menu in Figure 4-2. The `'>'` character after the `Restrict` operation is used to indicate that more operations are available, and ABF provides mechanisms for viewing them.

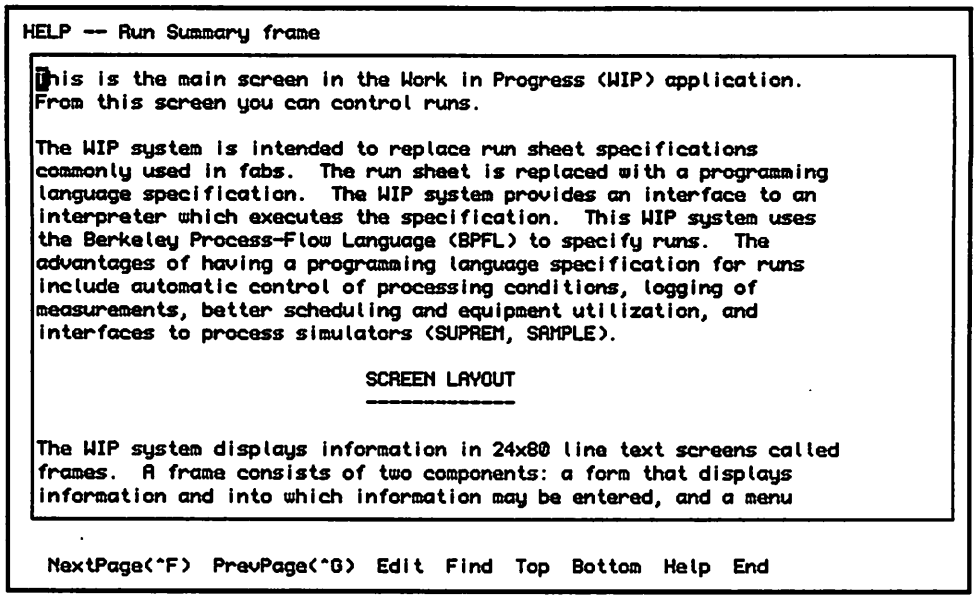

Figure 4-3: Run-Summary help.

tion. `Defaults` allows the user to set defaults for the UI process that control system behavior (e.g., the user can decide which editor to use for process flows). `Detail` provides more information about a run. `WIP-Log` displays the WIP-log for a run as described in section 4.3. `Restrict` lets the user establish which runs to display in the `Run-Summary` frame. `Version` is used to access the version-control system described in section 4.5. `Quit` causes the UI process to terminate.

4.2 Starting and Controlling a Run

This section describes how a run is created and controlled. When the `Create` operation in the `Run-Summary` frame is selected, the `Create-Run` frame shown in Figure 4-4 is displayed. The operations available at this frame are `Help`, `List-PFlows`, `Start-Run` and `End`. The `List-PFlows` operation displays a list of the approved process flows. `Start-Run` creates a new run. This operation causes the WIP interpreter to begin execution of the process flow. `End` returns to the `Run-Summary` frame.

To start a run, the user enters a run name, a process flow, a mask set and a lot size. The process flow is specified by a name and a version. In this example, the user has selected version 1.1 of the `cmos-16` process flow and named the run "`cmos test`." The initial steps of the process-flow code are shown in Figure 4-5. The flow has one argument named `implant-split`. As shown in Figure 4-4, the UI process fills in the default value of the argument. The user can modify the values of any argument if desired.

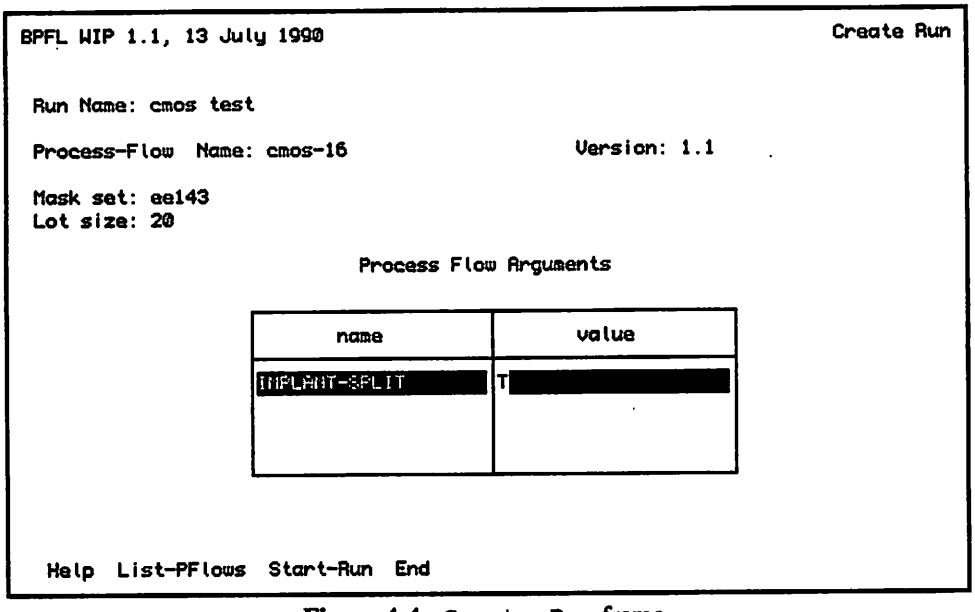

Figure 4-4: Create-Run frame.

After the Start-Run operation is executed by the user, the run will be initialized and the UI process displays the Run-Summary frame as shown in Figure 4-6. The new run appears on the bottom of the list of runs, with status of "starting," which indicates that the run is being ini-

```
defflow cmos-16(implant-split: = t)
  "U.C. Berkeley Generic CMOS Process (Ver. 1.6 14-April-89)
  (2 um, N-well, single poly-Si, single metal)"
begin
  step ALLOCATE-WAFERS do
    let spec := bare-silicon-wafer(crystal-face: 100,
                                   resistivity: [{18 ohm-cm}, {22 ohm-cm}]
                                   quality: 'product, dope: 'p);
    begin
      allocate-lot(names: '(cmos, nwell, nch),
                   sizes: list(*product-lot-size*, 1, 1),
                   snapshot: spec);
    end;
    /* Wafers in the cmos lot are product wafers */
    lot('product) := lot('cmos);
    with-lot 'nwell do
        measure-bulk-resistivity(tag: "initial");
    end;
  end;

  with-lot 'cmos do
    step WELL-FORMATION do
      step INIT-OX do
        wet-oxidation(time: {11 min}, temperature: {1000 degC},
                      target-thickness: {1000 angstrom});
        pattern(mask-name: 'NWELL);
      end;
```

Figure 4-5: CMOS-16 version 1.1 process flow code.

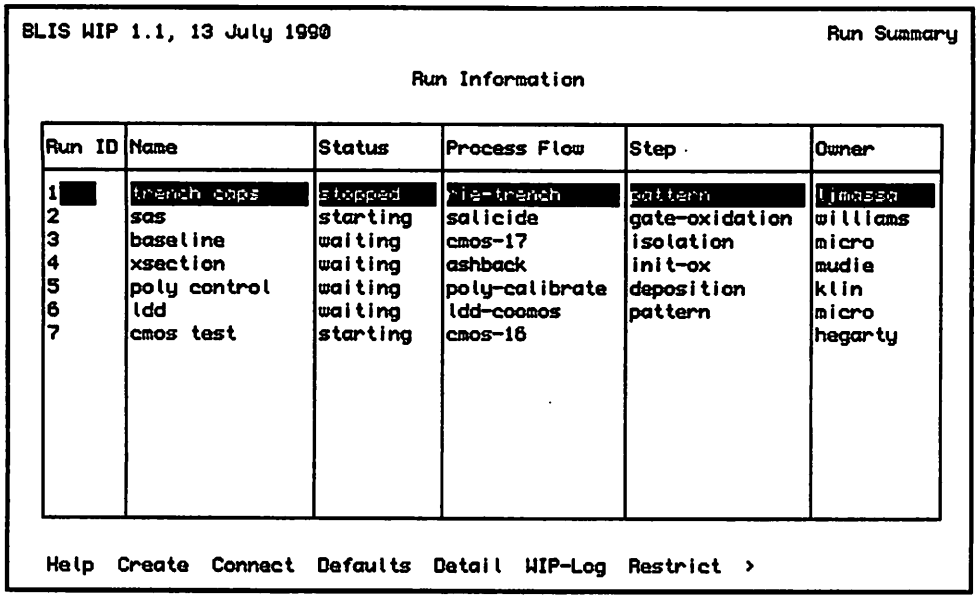

Figure 4-6: Run-Summary after creation of new run.

tialized. Once the run is initialized, the process-flow code begins to execute. The first processing operation allocates wafers:

```
allocate-lot(names: '(cmos, nwell, nch),
             sizes: list(*product-lot-size*, 1, 1),
             snapshot: spec);
```

When this step is reached, the WIP interpreter executes a `user-dialog` operation, which sends a request to the UI process that indicates that the run requires attention. The UI process displays a dialog box toward the top of the screen as shown in Figure 4-7 which indicates the name of the run

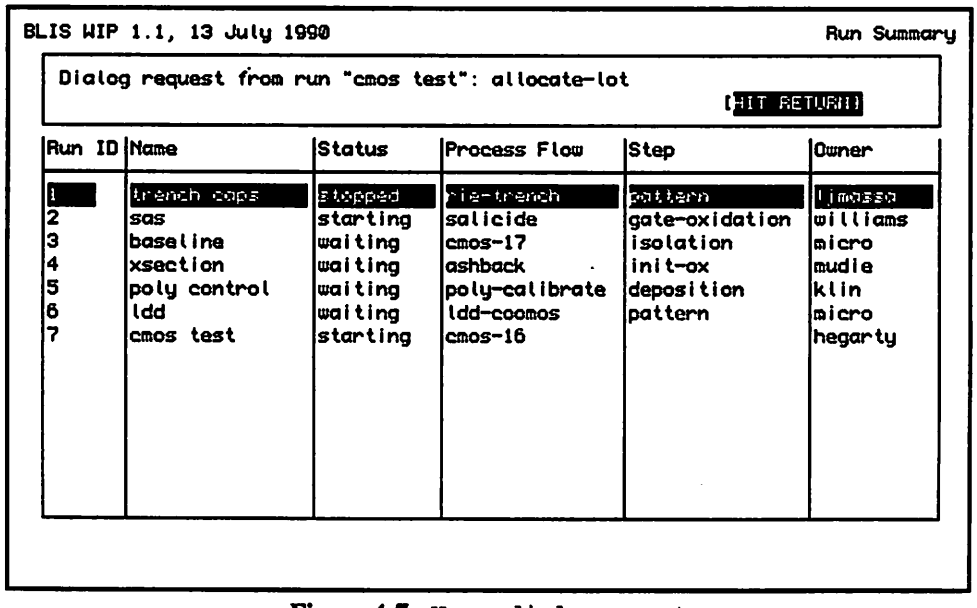

Figure 4-7: User-dialog request.

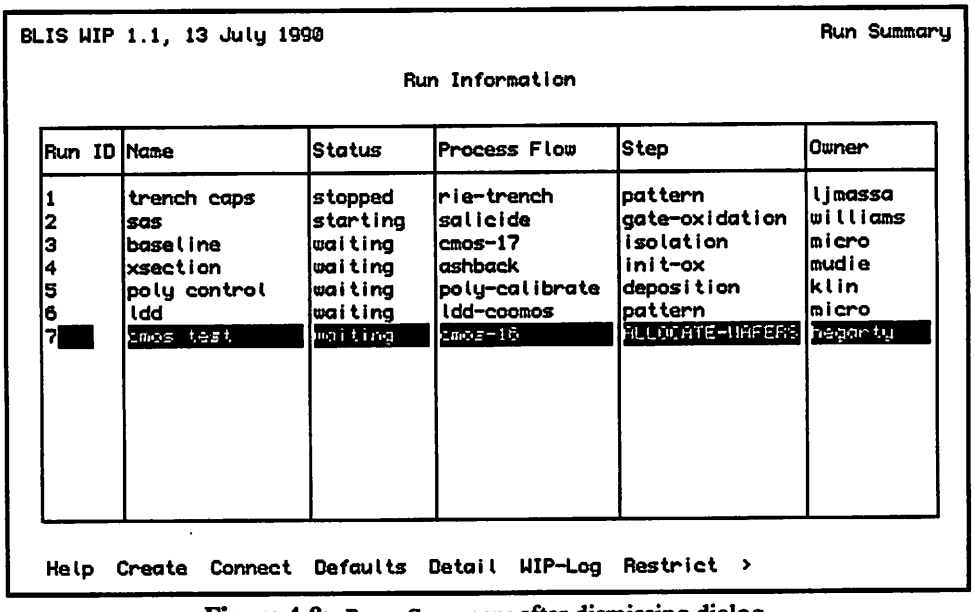

Figure 4-8: Run-Summary after dismissing dialog.

and a brief message describing the request. The dialog box is dismissed by hitting the return key, and the run information for "cmos test" is updated to reflect the current status of the run (i.e., waiting), as shown in Figure 4-8. A run in the state "waiting" is suspended pending a response from the user or equipment. The user can connect to the run so that he or she can respond to the request.

The user-dialog frame invoked by the run is displayed to the user when he or she connects to the run. In this example, the Allocate-Lot frame shown in Figure 4-9 is displayed. All user-

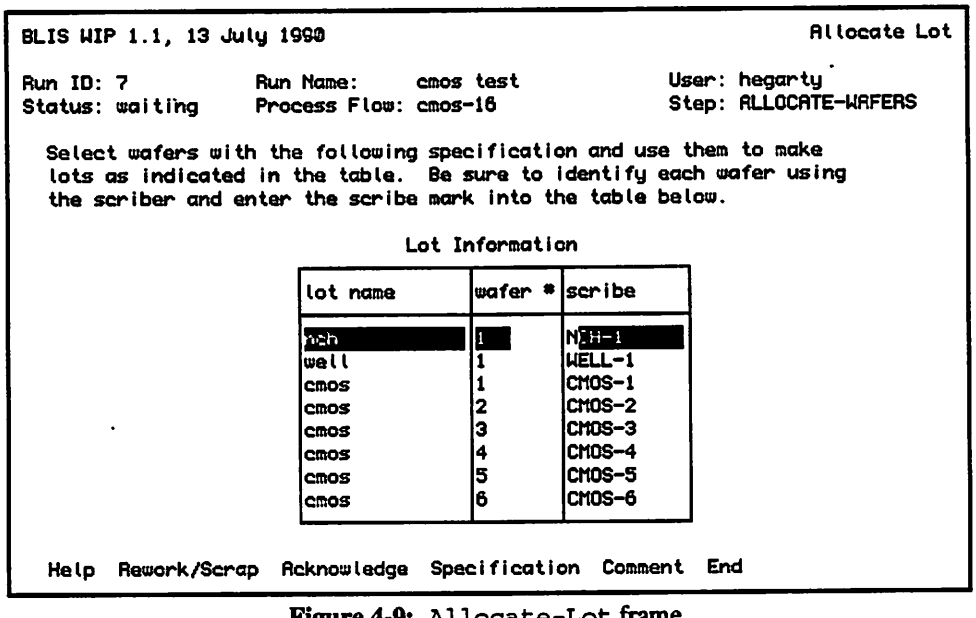

Figure 4-9: Allocate-Lot frame.

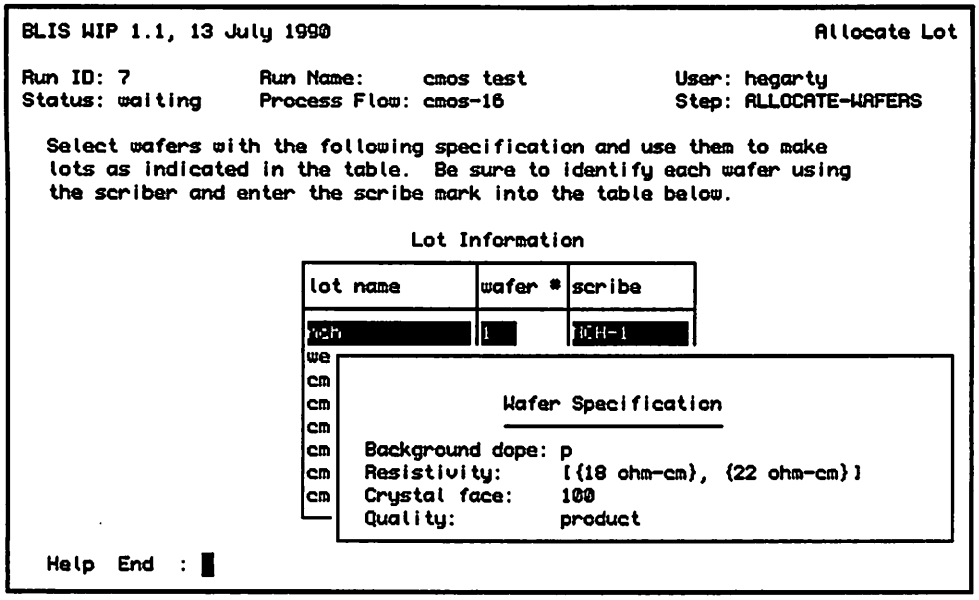

Figure 4-10: `Wafer-Specification` form.

dialog frames display information about the run in two lines at the top of the frame and they support the operations listed in Table 4-2. A user-dialog frame may also support additional operations (e.g., the `Specification` operation in the `Allocate-Lot` frame shown in Figure 4-9).

The `Allocate-Lot` frame instructs the user to allocate and name (or scribe) wafers. The system chooses default scribe names by appending the wafer number within a lot to the name of the lot (e.g., so the second wafer in lot `cmos` is named `CMOS-2`). The specification of the wafers to be allocated may be displayed in a pop-up form by the `Specification` operation, as shown in Figure 4-10. Once the user has scribed the wafers and entered the names into the `Lot-Information` table, the `Acknowledge` operation is executed, which completes the user-dialog, passes the results back to the WIP interpreter, and the run proceeds.

The next user-dialog operation in the process flow occurs in the `measure-bulk-resistivity` procedure. The code for `measure-bulk-resistivity` is shown in Figure 4-11. In the following discussion, the code in Figure 4-11 will be referred to by the line numbers on the

Operation	Description
`Help`	Displays help screen for the frame.
`Rework/Scrap`	Force rework or scrap wafers.
`Acknowledge`	Respond to the dialog.
`Comment`	Attach a comment to the dialog.
`End`	Return without responding to the dialog.

Table 4-2: User-dialog frame operations

left-hand side of the figure. Lines 2–6 query the wafer-state model to extract the nominal bulk resistivity specified in the model, and this value is passed to `user-dialog`. Lines 10–15 calculate a range attribute that is used to check values entered by the user. The `user-dialog` procedure call is:

```
results := user-dialog('sonogage, tag: tag, nominal: nominal,
                       limits: limits, wafer-id: wafer-id(wafer));
```

The first argument to `user-dialog` is the name of the frame to display (i.e., `Sonogage`). The `tag` argument is used to tag the WIP log-object created by the `user-dialog`. The rest of the arguments are passed to the `Sonogage` frame. Different frames take different arguments. The `Sonogage` frame takes three arguments, `nominal`, `limits` and `wafer-id`. `Nominal` specifies the expected value of the measurements, `limits` specifies the acceptable range of measurements, and `wafer-id` specifies the identifier for the wafer on which the measurements are to be performed.

When the user-dialog is executed and the user connects to the run, the `Sonogage` frame shown in Figure 4-12 is displayed. Note that the `nominal` argument is displayed to the user, telling the user approximately what measurements to expect. The `scribe` of the wafer with identifier `wafer-id` is displayed so that the user knows what wafer to use. The frame is described in more de-

```
 1 defflow measure-bulk-resistivity(tag)
 2 let wafer := pick-test-wafer();
 3     ss := wafer-snapshot(wafer);
 4     seg := first(find-segments(ss, material: #m(substrate));
 5     mat := pif-attr-val(seg, :material, ss);
 6     nominal := material-attr(mat, :resistivity);
 7     limits := nil;
 8     results := nil;
 9 begin
10   if interval-p(nominal) then
11      limits := make-interval(interval-min(nominal) * 0.5,
12                              interval-max(nominal) * 2.0)
13   else
14     limits := make-interval(nominal * 0.5, nominal * 2.0);
15   end;
16   results := user-dialog('sonogage, tag: tag, nominal: nominal,
17                          limits: limits, wafer-id: wafer-id(wafer));
18   getf(results, :average) := sigfigs(average(getf(results, :measurements)),3);
19   wip-log('sonogage, results);
20   getf(results,:average);
21 end;
```

Figure 4-11: `Measure-bulk-resistivity` definition.

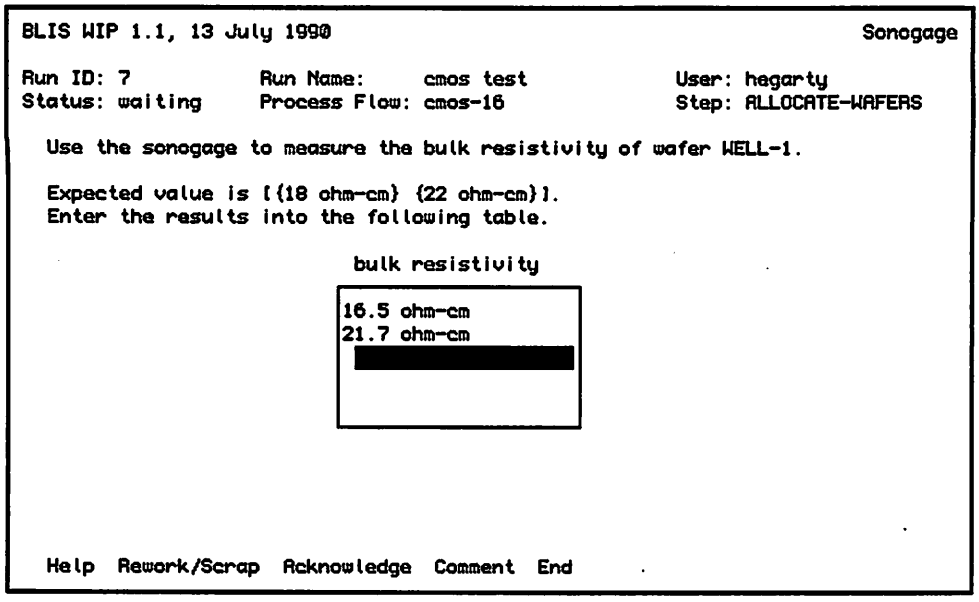

Figure 4-12: Sonogage frame.

tail in Chapter 5. Figure 4-12 shows the frame as it appears after the user has entered two measured values.

The UI process performs edit checks on measurements entered by the user. For example, the only acceptable units for bulk-resistivity are units with the same dimensions as `ohm-cm`. If a user enters a value with different dimensions, an error is signalled. Figure 4-13 shows the error message that results if the user enters a value with inconsistent dimensions. Values like `{300 Mohm-km}` are acceptable, because the system parses SI[1] unit specifications.

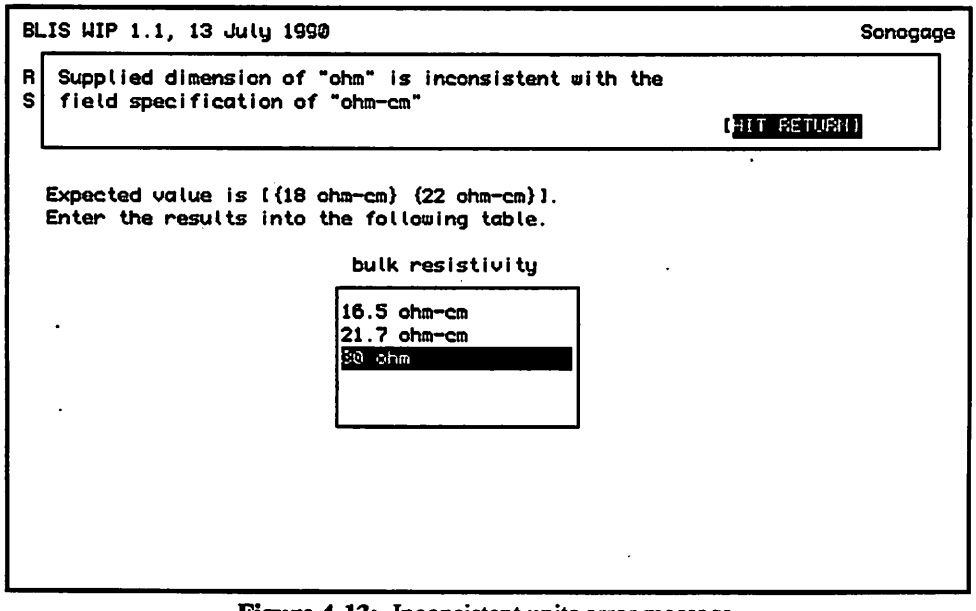

Figure 4-13: Inconsistent units error message.

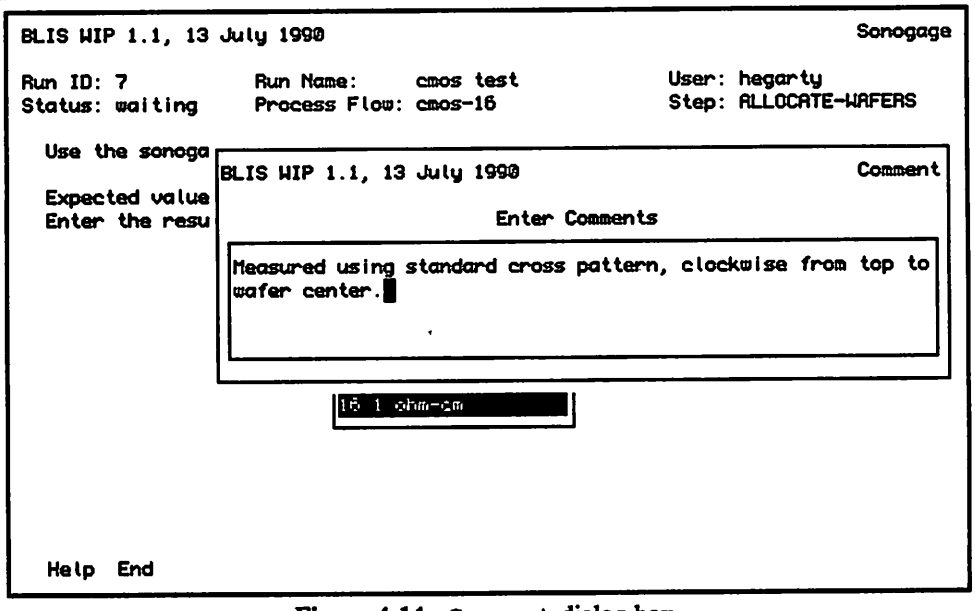

Figure 4-14: Comment dialog box.

The run management system can also warn the user if an entered value is outside an allowable range. Since the code in Figure 4-11 above passes a `limits` attribute to the `user-dialog`, values outside the range specified generate a warning message.

The user may also attach comments to any user-dialog interaction. For example, if the user wants to make a note of the measurement pattern used to measure bulk resistivity, a comment like the one shown in Figure 4-14 can be added by selecting the `Comment` operation. This comment is attached to the log record associated with the frame when the frame is acknowledged. A run typically consists of more than 100 individual user-dialog operations like the examples above.

Consider again the `Run-Summary` frame operations listed in Table 4-1. The `Restrict` operation is used to set up what runs to display in the `Run-Summary` frame. For example, this operation can be used to display only runs belonging to a particular owner or running a certain process-flow. Figure 4-15 shows the `Restrict` pop-up dialog displayed by the `Restrict` operation. In this case, the user has entered restrictions to display only runs using the `cmos-16` process flow and owned by `hegarty`.

The `Detail` operation displays the `Run-Detail` frame, which provides more information about run state, and allows the user to modify the run. Figure 4-16 shows more detail for the run created in the above example. The top two lines of the frame provide basic information about

[1] Système International, the International System of Units.

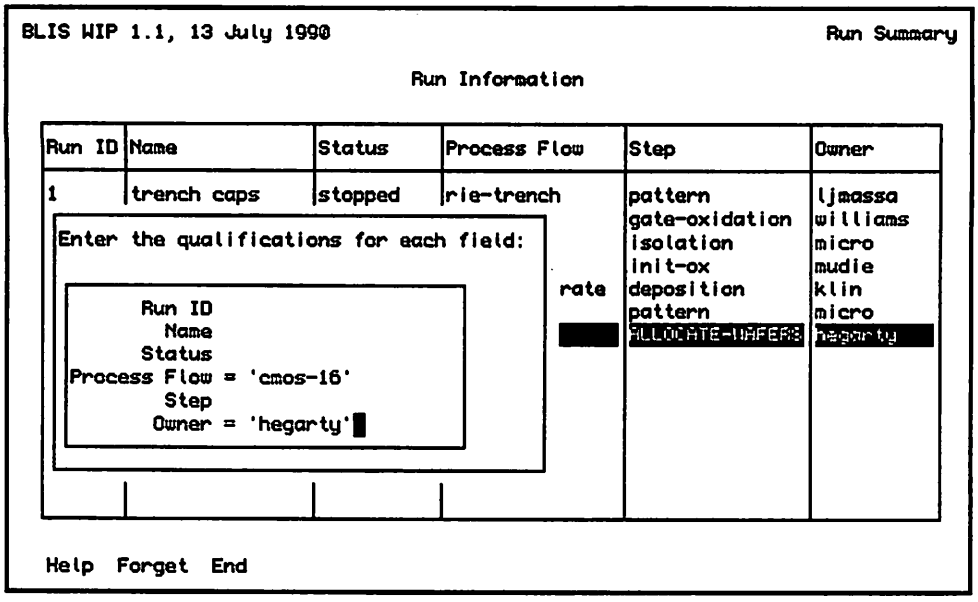

Figure 4-15: `Restrict` dialog box.

run status: the run `step-path`, the process flow, the mask-set, and the lot-size. The `Lots` table on the left displays the names of all lots in the run. The system defines additional lots: `rework`, `scrap`, `product` and `*all-wafers*`. The `scrap` and `rework` lots are discussed in chapter 2. The `product` lot specifies the wafers that contain product, and the `*all-wafers*` lot contains all active wafers in the run (i.e., all wafers that have not been scrapped). The table on the right displays information about the wafers in the lot selected in the first table. In Figure 4-16, the `*all-wafers*` lot is selected.

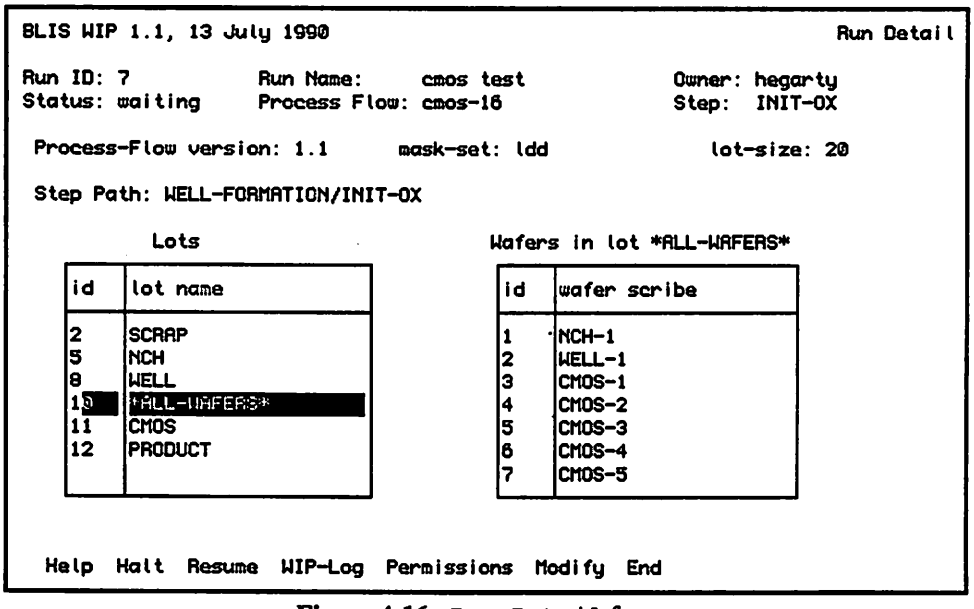

Figure 4-16: `Run-Detail` frame.

Operation	Description
Help	Displays help screen for the frame.
Halt	Stop the run and save the current state.
Resume	Restart the run after stopping it.
WIP-Log	Display the WIP-Log for this run.
Permissions	Display and update run permissions.
Modify	Modify the run lots or the process flow code.
End	Return to the run-summary frame.

Table 4-3: Run-Detail frame operations.

The operations available from the Run-Detail frame are listed in Table 4-3. Halt stops a run and Resume restarts a stopped run. WIP-Log displays the WIP-log for the run as described in the next section. Permissions lets the run owner specify which users are allowed to connect to the run using the Run-Permissions frame shown in Figure 4-17. Modify lets the user change an active run and is described in section 4.4.

4.3 Browsing Processing History

The WIP system maintains a record of all events that occur during processing in the WIP log. Events recorded in the log include user-dialogs, equipment operations, and error or warning messages, as well as log object records explicitly written by a BPFL procedure.

The UI process allows a user to browse the WIP log of a particular run or group of runs. Selecting the WIP-Log operation from the Run-Detail frame (Figure 4-16) displays the WIP

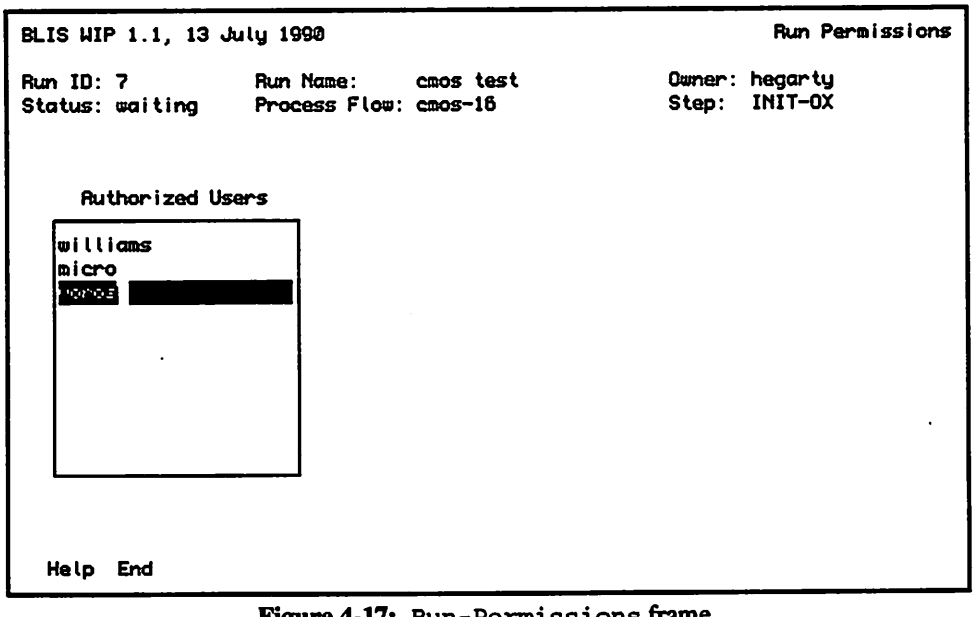

Figure 4-17: Run-Permissions frame.

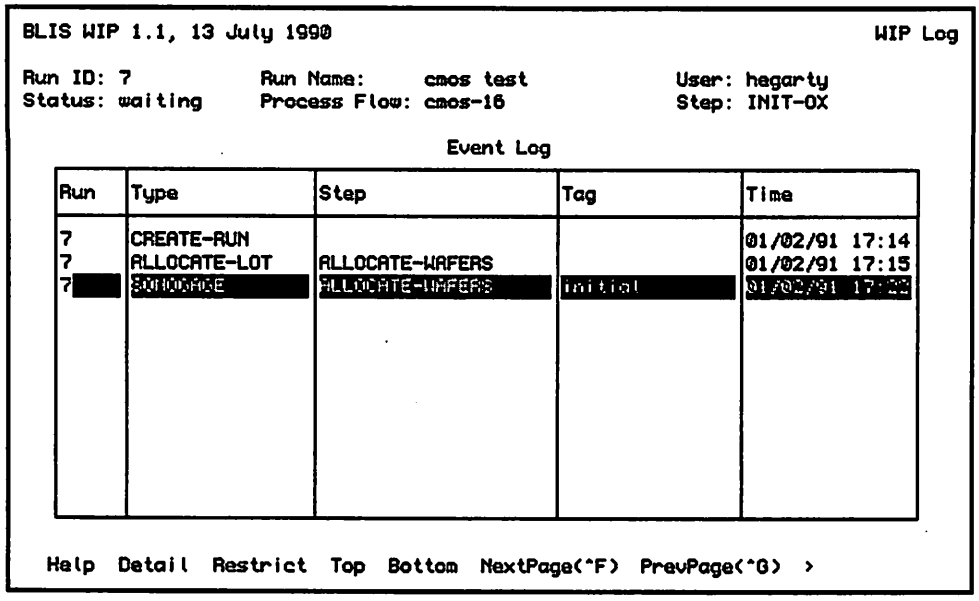

Figure 4-18: WIP-Log frame.

log for a run as shown in Figure 4-18. The WIP log for this particular run has three events recorded in it.

The operations provided in the WIP-log frame are shown in Table 4-4. The Detail operation shows more information about the selected event. The Restrict operation can be used to enter criteria for the display of events (e.g., only Sonogage events could be displayed). Top, Bottom, NextPage and PrevPage are used to move around within the events displayed in the table.

Figure 4-19 shows the Sonogage-Log frame displayed by choosing the Detail operation after selecting the Sonogage event in Figure 4-18.[1] The top two lines of the frame display information about the run. The two lines below that display information about the WIP log record

Operation	Description
Help	Displays help screen for the frame.
Detail	Display detail for the selected event.
Restrict	Enter criteria restricting display of events (e.g., display only events of a certain type).
Top	Move to top of table.
Bottom	Move to bottom of table.
NextPage	Move back one page in table.
PrevPage	Move forward one page in table.
End	Return to the calling frame.

Table 4-4: WIP-Log frame operations.

[1] An event is selected by positioning the cursor on the corresponding row in the table.

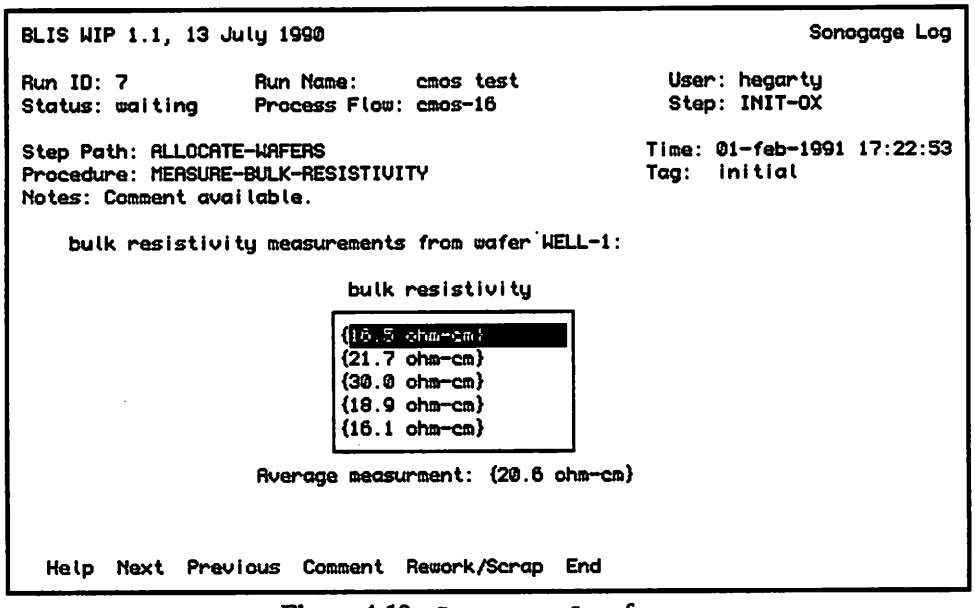

Figure 4-19: Sonogage-Log frame.

being examined: the step-path and time at which the event was recorded, and the procedure which wrote the event. A tag string may also be attached to an event, which simplifies queries to retrieve log records. The tag on this event is "`initial`," as seen in the code in Figure 4-5.

The code that wrote this log record (i.e., the `measure-bulk-resistivity` procedure in Figure 4-11) calculated the average values of the sonogage measurements and logged the results with the code in lines 18–19:

```
getf(results, :average) := sigfigs(average(getf(results, :measurements)),3);
wip-log('sonogage, results);
```

The first line of code calculates the average value (to three significant figures) of the `measurements` attribute of the values returned by the `user-dialog` operation. This value is then added to the results as an `average` attribute, and the results are recorded in the database using the `WIP-log` procedure. The average value is displayed in the `Sonogage-Log` frame below the table of measurements.

Operation	Description
`Help`	Displays help screen for the frame.
`Next`	Display the next event.
`Previous`	Display the previous event.
`Comment`	Display comment for this event (if any).
`Rework/Scrap`	Display rework or scrap initiated at this event (if any).
`End`	Return to the `WIP-Log` frame.

Table 4-5: Log frame operations.

Every log frame supports the operations listed in Table 4-5. Additional operations are available from some log frames (e.g., the `Allocate-Lot-Log` frame has a `Specification` operation that is used to display the specification for the allocated wafers). Each log frame has a field which displays special messages about the event shown in the frame. For example, the event displayed in the frame in Figure 4-19 has a comment recorded with it, and this information is reported to the user.

4.4 Dynamically modifying a run

The WIP system allows runs to be modified while they are executing. A user can add or remove wafers, import wafers from another run, split a run into multiple runs, and modify the process-flow code used by a run. The `Modify` operation in the `Run-Detail` frame is used to modify a run. When this operation is executed, a submenu of four modify operations is displayed. The operations are:

1. `Modify-Lots`,
2. `Split-Run`,
3. `Change-Flow`, and
4. `Import-Wafers`.

Each of these operations is described in detail below.

Wafers can be moved between lots and removed from a run by using the `Modify-Lots` frame shown in Figure 4-20. Table 4-6 lists the operations in the `Modify-Lots` frame. Wafers can be removed from a lot or moved to another lot. In this example, wafers from the `split-low` lot of the `baseline` run are to be moved to the `split-med` lot. A new lot can be created by typing in a `lot-name` that does not currently exist.

Operation	Description
`Help`	Displays help screen for the frame.
`Remove`	Remove the selected wafer from the lot.
`Add`	Add the selected wafer to the other lot.
`New`	Create new wafers for the run.
`Change-Scribe`	Type in a new wafer scribe for the selected wafer.
`End`	Return to the `Run-Detail` frame.

Table 4-6: `Modify-Lots` operations.

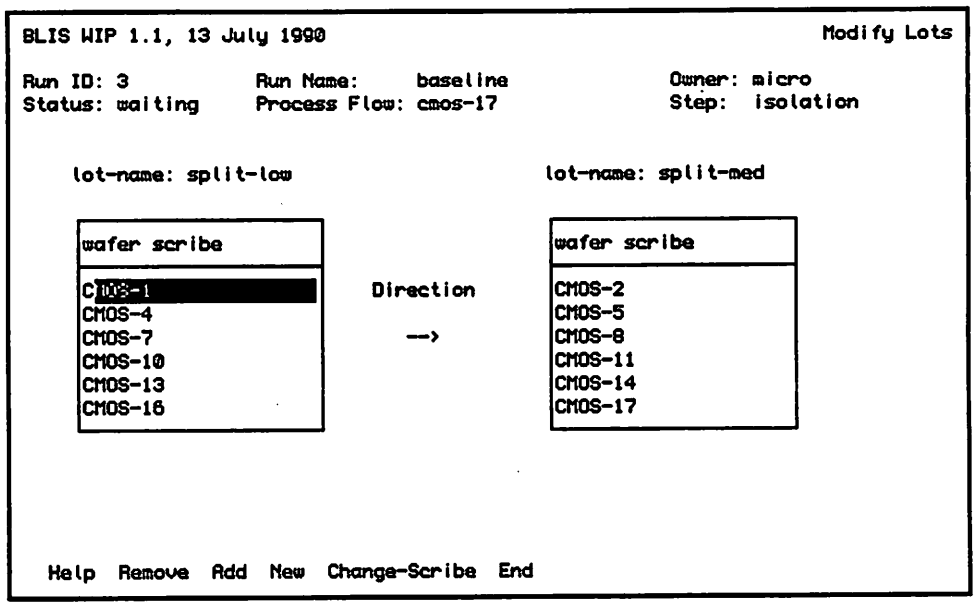

Figure 4-20: Modify-Lots frame.

The New operation allocates new wafers for a run. This operation calls the New-Wafers frame shown in Figure 4-21. As can be seen in Figure 4-21, the user must specify to which lot to add the wafers, how many wafers to create, and the BPFL code that returns the snapshot describing the new wafers. The user specifies a process-flow name and version, and the code to call within that flow to generate the snapshot. New wafers may also be added to a run with the allocate-lot procedure, but the ability to add wafers without altering code is useful when test wafers have been damaged and new test wafers are required.

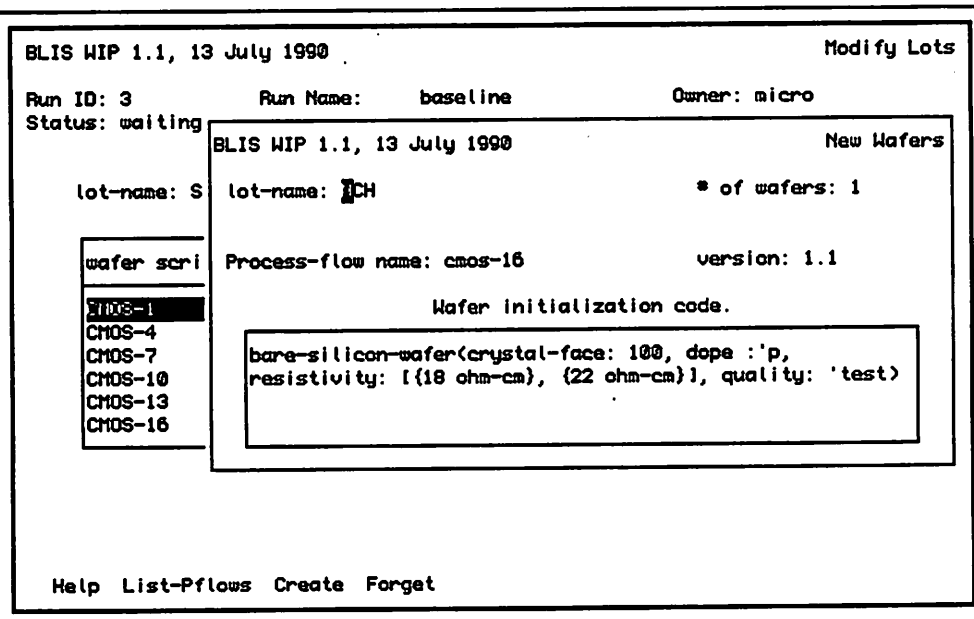

Figure 4-21: New-Wafers frame.

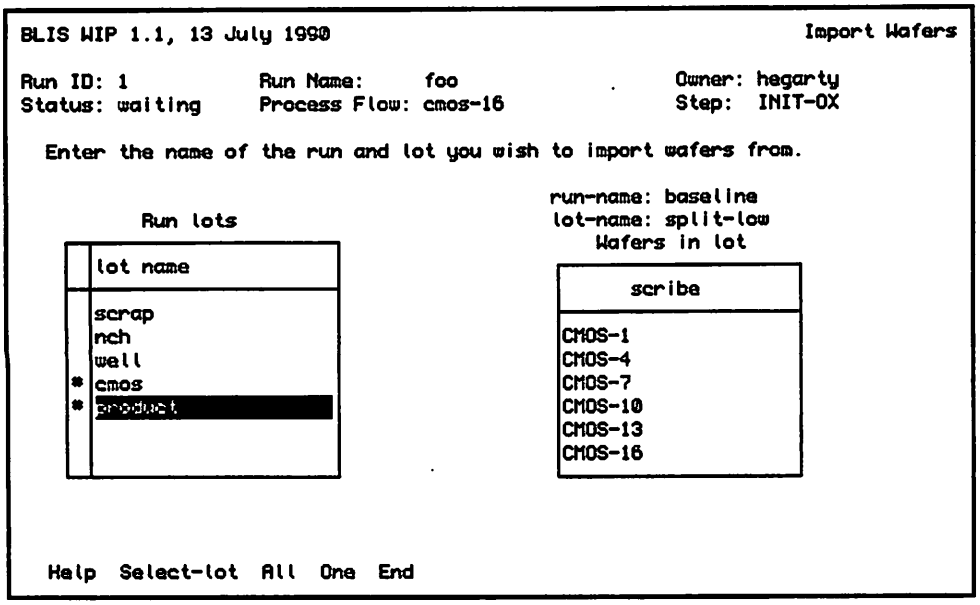

Figure 4-22: `Import-Wafers` frame.

Wafers can be moved between runs using the `Import-Wafers` frame shown in Figure 4-22. The `Select-Lot` operation is used to choose the lots in which to place imported wafers. In this example, the `cmos` and `product` lots of the run created in section 4.2 have been selected. The `run-name` and `lot-name` fields are used to select a lot from another run. The wafers in that lot are displayed in the `Wafers-in-lot` field. In this example, the `split-low` lot of run `baseline` is displayed. The `All` operation is used to import all wafers from the selected lot, and the `One` operation is used to import the currently selected wafer in the table. A graphical user-interface would specify these actions by pointing at the wafer with the mouse and moving it to an icon that represented the lot.

A run may be split into multiple runs with the `Split-Run` frame shown in Figure 4-23. This operation is useful for experimenting with different treatments on wafers that have undergone identical processing prior to the run split. In this example, the `baseline` run is being split into three runs, each of which will receive a different implant dose. When a run is split, the old run is halted and the new runs are created with the same lot names as the old run but with no wafers in any of the lots. Wafers must be moved into the new runs from the original run with the `Import-Wafers` frame before the new runs are started.

BPFL process flows may be altered while a run is executing. For example, updates to standard library procedures (e.g., `measure-oxide-thickness`) normally should be incorporated

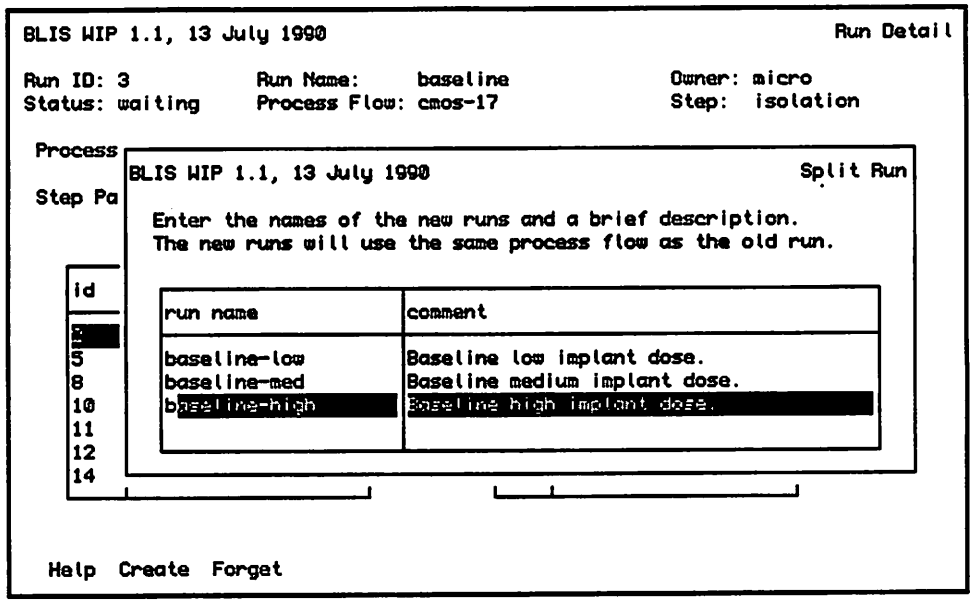

Figure 4-23: `Split-Run` frame.

into a flow immediately since library code enforces facility policy. On the other hand, some changes to process-flow code may be impossible to use in an existing run because the run is executing a section of code between changes to the process-flow which could result in run-time errors. The desired response of the run to changes in its process-flow code are specified using the `Modify-Flow` frame shown in Figure 4-24. The `Process-Flow` in this case is version 1.1 of `cmos-16`. The `action` field specifies how the run responds to changes in code. This field can have one of three values:

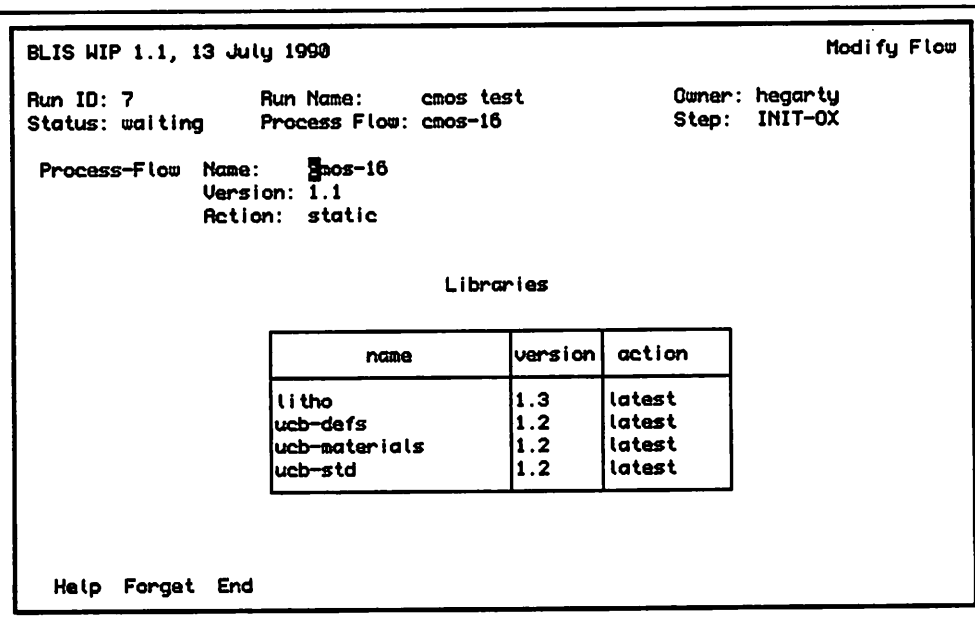

Figure 4-24: `Modify-Flow` frame.

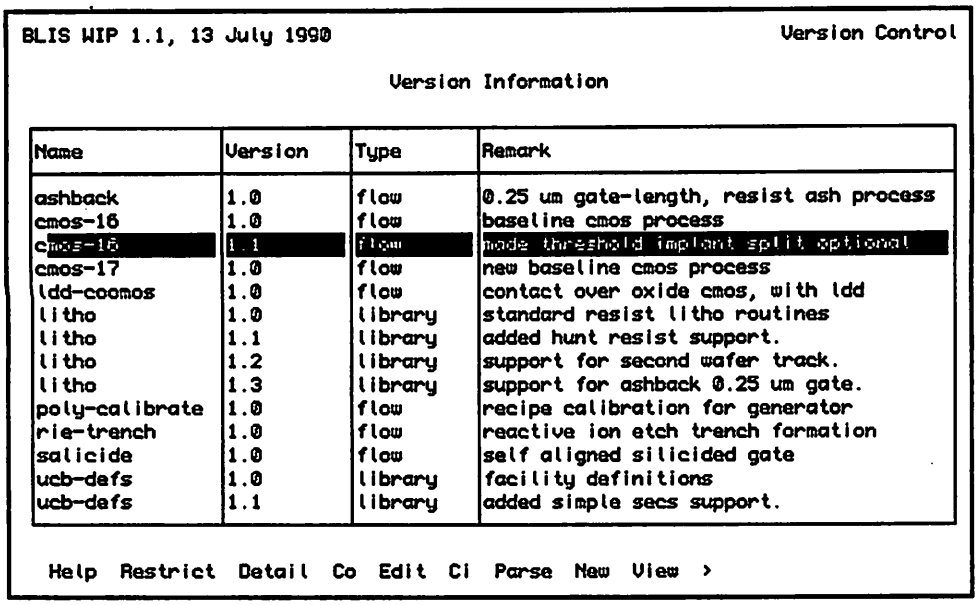

Figure 4-25: `Version-Control` frame.

1. `Static` – The process flow used by the run is never updated,
2. `Latest` – The process flow used by the run is always updated to the latest version available, or
3. `Query` - Whenever a new version of a process flow is created, the run owner is asked whether or not to use the new version.

In this example, the `action` field has the value '`static.`' If later versions of `cmos-16` become available while the run is executing, they will not be used.

BPFL process flows can use standard libraries of procedures using the `requires` declaration. In Figure 4-24, four libraries used by the "`cmos test`" run are shown in the `Libraries` table. The default action for libraries is `latest`. The next section describes how flows and libraries are created and updated.

4.5 Version control

BPFL supports *flows*, which contain the top-level code for processing wafers (e.g., `cmos-16`), and *libraries*, which contain standard procedures shared between flows (e.g., `measure-bulk-resistivity`). The term *module* is used to refer to both flows and libraries. BPFL code may be created and edited using the `Version-Control` frame shown in Figure 4-25. The operations provided in the frame are described in Table 4-7. The WIP system uses the Revision Control System (RCS) to organize and maintain different version of BPFL code [51]. Whenever a user

```
/* bpfl process flow
   $Log:        cmos-16.b,v $
  Revision 1.1  91/01/13  08:31:07  hegarty
  added implant-split operations

  Revision 1.0  90/10/26  14:40:42  hegarty
  Initial revision

/*
require(cmos-lib, version: latest);

defflow cmos-16(implant-split: = t)
  "U.C. Berkeley Generic CMOS Process (Ver. 1.6 14-April-89)
  (2 um, N-well, single poly-Si, single metal)"
begin
  step ALLOCATE-WAFERS do
    let spec := bare-silicon-wafer(crystal-face: 100,
                                   resistivity: [(18 ohm-cm) (22 ohm-cm)]
                                   quality: 'product, dope: 'p);
    begin
      allocate-lot(names: '(cmos, nwell, nch),
                   sizes: list(*product-lot-size*, 1, 1),
-----Emacs: cmos-16.b              (Common Lisp)----Top----------------------
package is "BPFL"
```

Figure 4-26: Editing a process flow.

wishes to modify a module, he or she may *check out* a module.[1] Only one person may have a particular module checked out at any given time. The user can edit the module, run it, and debug it until satisfied that it is ready for use by others. The flow is then *checked in* and assigned a version number. RCS locks flows to prevent simultaneous modification of the same version, and stores the code modification tree in an efficient way.

As an example of the use of the version control system, if the user checks out version 1.1 of `cmos-16` and selects the `Edit` operation, the user's default editor (e.g., `emacs`) is invoked on the flow as shown in Figure 4-26. The top few lines of the file list RCS information maintained

Operation	Description
`Help`	Displays help screen for the frame.
`Restrict`	Enter criteria restricting display.
`Detail`	Display further detail about the selected flow or library.
`Co`	Check out a flow or library.
`Edit`	Edit BPFL code
`Ci`	Check in a flow or library.
`Parse`	Parse BPFL code and check syntax.
`New`	Create a new flow or library.
`View`	Examine code without making changes.
`Update-Runs`	Update code used by runs.
`End`	Return to the `run-summary` frame.

Table 4-7: `Version-Control` operations.

[1] Each flow has a set of users authorized to run and modify it.

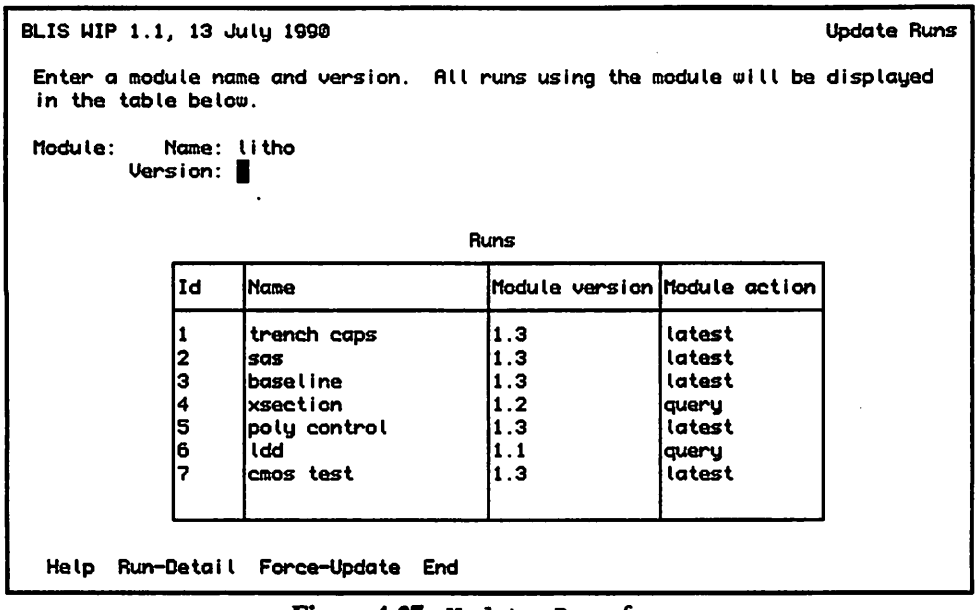

Figure 4-27: `Update-Runs` frame.

about the flow. The `require` statement indicates that this process-flow will use the latest version of the the `ucb-std` library. The definition of the `cmos-16` process flow appears in the bottom half of the screen. At this point, the user can modify the code and check it using the `Parse` operation. When the user is finished modifying the code, it can be checked in.

Users can set up the actions for runs using the `Modify-Flow` frame shown in Figure 4-24. A lab manager may wish to force all runs to be updated to use a new version of a module. This is accomplished using the `Update-Runs` operation. When this operation is selected, the `Update-Runs` frame in Figure 4-27 is displayed. The user types in a module name and the runs using that module are displayed in the `Runs` table. The user may also type in a module `version` if it is desired to restrict updates to runs using only a particular version of the module. The `Run-Detail` operation can be used to display more information about a selected run. `Force-Update` can be used to force the selected run to use a new version of the module. Only a user with WIP root permission is able to execute `Force-Update` on runs owned by other users.

4.6 Summary

The BPFL WIP system is designed to permit the use of BPFL process flows in a fabrication environment. It is composed of many processes that communicate with users, equipment, and the CIM database. Runs may be started, executed and modified by a user with a forms-based user-interface process. The WIP system maintains a log of all events that occur while processing a run,

including user-dialogs and error or warning messages. The system uses RCS to organize and maintain different versions of BPFL code.

[This page intentionally blank]

Chapter 5
Implementation

This chapter describes the implementation of the WIP system. First, the process architecture and interprocess communication channels between the processes are described. Second, the WIP database design is presented. Next, the operation of the User-Interface process is explained, and the method of defining user frames is described. Fourth, the WIP interpreter process is discussed, including the methods used to execute BPFL code and save run state in the database. Finally, the run management system is described and the way runs are modified is discussed.

5.1 Processes and Interprocess Communication

The WIP system is composed of three types of processes:

1. the WIP interpreter process,
2. the User-Interface process (UI process), and
3. the Equipment Interface process (EI process).

These processes are shown in Figure 5-1. This section describes each process and the communication between them.

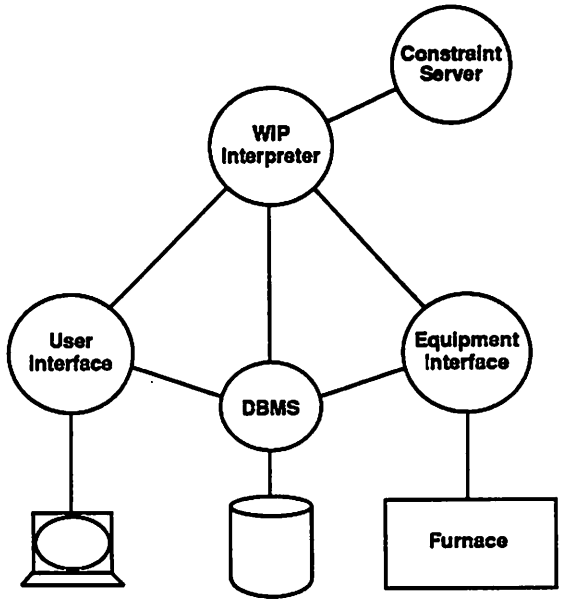

Figure 5-1: WIP system architecture.

The WIP interpreter process executes runs. A run corresponds to an execution of a BPFL process flow. Since the WIP interpreter process is a large Common Lisp program, the process can execute several runs concurrently. In other words, the WIP interpreter process is a server process. In order to accomplish this, the WIP interpreter process maintains data structures that describe the run state (e.g., the next statement to execute, the names and values of local variables created by the program, and data retrieved from the database) for each run.

The WIP interpreter process uses subsidiary processes to implement constraints. For example, there is a timing constraint server that is responsible for alerting the WIP interpreter process when a timing constraint expires. In addition, the WIP interpreter process coordinates the activities of the EI and UI processes.

The WIP interpreter process maintains the state of runs in the database to provide fault-tolerance. Fault tolerance is necessary in a WIP system because semiconductor process flows take weeks or months to execute, and it is likely that a computer system failure will occur before a run is complete. Consequently, all state information about a process must be saved in non-volatile storage so that a run can be restarted from its last saved position when the system crashes. The WIP interpreter saves the run-state data structures to the database. State is saved whenever a run is stopped (e.g., while waiting for user input).

The UI process is the user-interface to runs. The UI process reads information about the state of a run from the database and displays it to the user. Each active user has a UI process, and the user can respond to dialogs, examine run state and connect to different runs (i.e., the interface shown in chapter 4). The UI process also writes events (e.g., user-dialog events) to the WIP-log in the database and is responsible for enforcing access control to runs. For example, it must prevent multiple users from simultaneously connecting to the same run, and it must prevent users from connecting to a run that they are not authorized to manipulate.

The EI process controls equipment and writes equipment status information into the database. The EI process is a Common Lisp implementation of the SECS[1] server developed by Wood and is described in more detail elsewhere [47].

---

[1] Semiconductor Equipment Manufacturers Institute (SEMI) Equipment Communications Standard protocol (SECS).

The processes communicate either through interprocess communication channels (IPC) or through the shared CIM database. Internet-domain connections (TCP/IP[1]) are used for real-time notification (e.g., a timing constraint expires). Non real-time communication is implemented by the database. For example, the sequence of operations leading to the display of a dialog-box to the user as a result of a `user-dialog` operation is as follows. When the WIP interpreter process interprets the `user-dialog` call, it writes a description of the user-dialog to the database. It then sends a TCP/IP message to the appropriate UI process for the run. The UI process receives the TCP/IP message and displays the specified dialog box. When the user instructs the UI process to connect to the run, the information stored in the database by the WIP interpreter process is read and used to display the user-dialog frame. When the user has filled in the user-dialog frame and is ready to continue the run, the UI process writes a WIP-log record to the database and sends a TCP/IP message to the WIP interpreter process to signal that the run is ready to proceed. The WIP interpreter process then resumes execution of the process flow for the run.

5.2 WIP Database

This section describes the data in the CIM database used by the WIP system. Figure 5-2 shows the *Entity-Relationship* (ER) diagram for the WIP database using Reiner's notation [52]. The entities described in the WIP database are:

1. Run – The state of a run.
2. Equipment – Processing equipment and utilities.
3. User – A person who uses the fab (e.g., equipment operators, administrators and technicians).
4. Wafer – A semiconductor wafer.
5. Snapshot – A PIF representation of wafer state.
6. Lot - A group of wafers.
7. Mask – A stepper photomask.
8. Mask set – A collection of masks for a circuit, one for each masking operation in a process flow.
9. Flow – A top level BPFL process flow.

[1] Transmission Control Protocol / Internet Protocol.

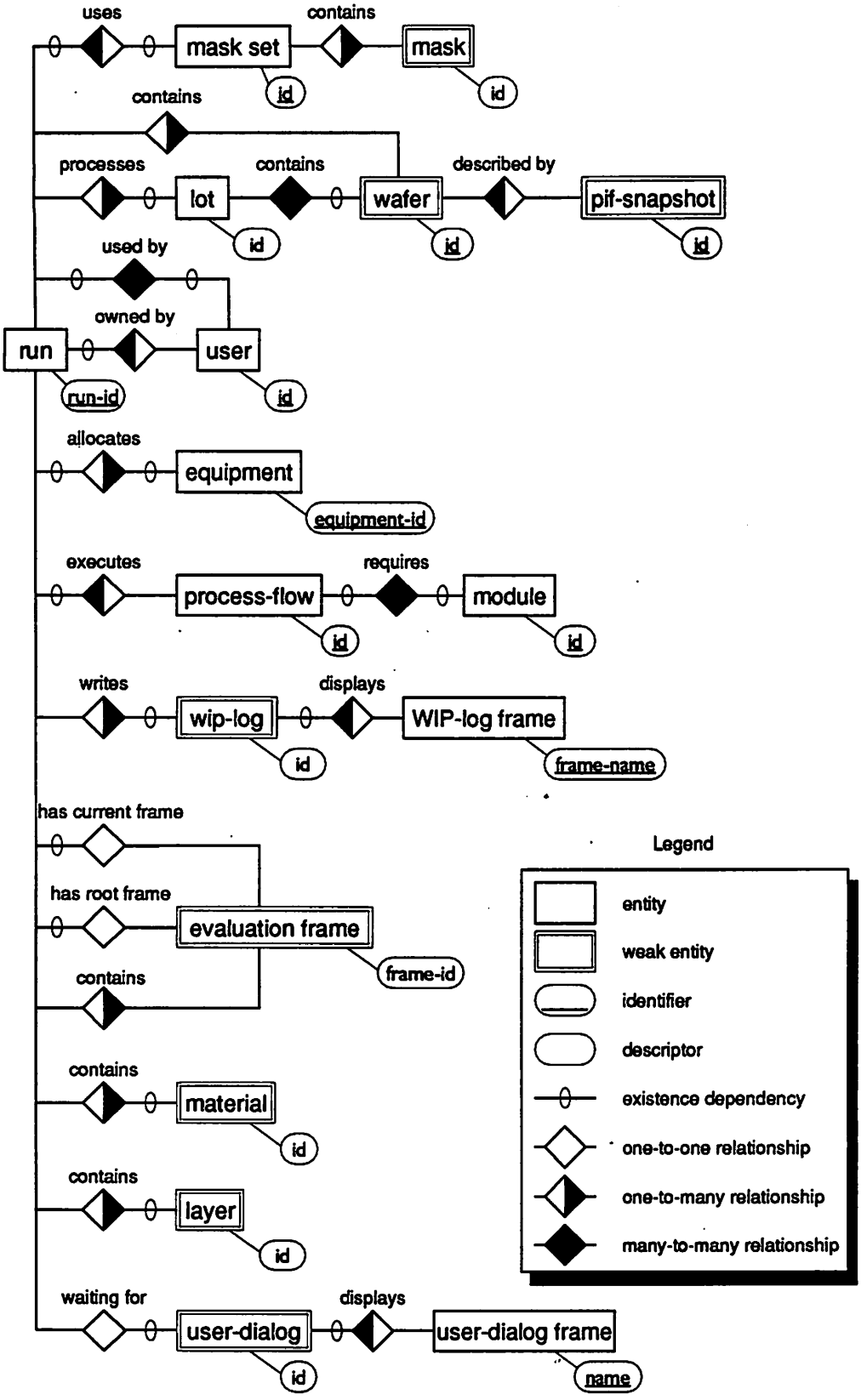

Figure 5-2: WIP database entity-relationship diagram.

10. Library – A library of BPFL procedures.
11. WIP-log – A WIP-log record.
12. WIP-log frame – An ABF frame used to display a WIP-log record.
13. Evaluation frame – A frame used to interpret BPFL code.
14. Material – A material description.
15. Layer – A PIF layer description.
16. User-dialog – A request from a run for input from a user.
17. User-dialog frame – A frame displayed by a run waiting for input.

The basic ER model consists of three classes of objects: entities, relationships and attributes. *Entities* are the principal data objects about which information is stored: for example, a `process-flow` is an entity, as shown in Figure 5-2. A particular occurrence of an entity is called an entity instance. For example, a particular process flow such as `cmos-16` is an instance of the `process-flow` entity.

Relationships represent associations among one or more entities. For example, a `run` is associated with the `process-flow` it executes. Relationships are described in terms of connectivity, role and existence.[1] The most common meaning associated with relationships is indicated by the connectivity between entities: one-to-one, one-to-many, and many-to-many. For example, a run is associated with one process flow but any process flow may be associated with many runs, so the connectivity between the `run` and `process-flow` entities is many-to-one.

A role is the function an entity plays in a relationship. For example, a run *executes* a process flow, so the role "executes" defines the function of the `run` entity in the relationship between `run` and `process-flow`. Roles are shown above relationships in Figure 5-2. The role describes the function that the entity on the right of the relationship plays with respect to the entity on the left of the relationship.

The existence of some entities depends on the existence of another entity. This is called *existence*. Existence of an entity in a relationship is defined as either mandatory or optional. If an instance of either the "one" or "many" side entity must exist for the entity to be included in the relationship, then it is mandatory. For example, the entity `run` may or may not be waiting for a

[1] Additional relationship meanings are defined by Reiner [52] but are not used here.

user-dialog, thus making the entity user-dialog in the "waiting for" relationship between run and user-dialog optional. However, if a user-dialog entity instance exists, a run instance must be waiting for that user-dialog, so the run entity is not optional in the relationship.

Attributes are characteristics of entities that provide descriptive detail about them. For example, a run entity has a name attribute. There are two types of attributes: identifiers and descriptors. An *identifier* (or key) uniquely determines an instance of an entity. A *descriptor* is used to specify a non-unique characteristic of a particular entity instance. For example, run-id is an identifier for a run (since every run has a unique id) but name is a descriptor. In the prototype implementation, identifiers are integers.

Entities have internal identifiers that uniquely determine the existence of entity instances, but *weak entities* derive their identity from the identity instances of one or more "parent" instances. For example, an evaluation-frame has no identifier, but derives its identity from the run that contains it.

The database used in the prototype WIP interpreter process is INGRES [53]. The SLING package written by Sedayao and Charness [54] is the application program interface (API) to the database system. SLING is an SQL API for Common Lisp. Because there is no way to directly write CLOS objects to INGRES, database tables are defined for each object type and CLOS methods are used to write objects to and read objects from the database. Appendix D contains the database table definitions for the entities shown in Figure 5-2. The WIP-log-frame and user-dialog-frame entities are created and maintained by ABF [50] and their definitions do not appear in the Appendix.

5.3 The User-Interface Process

The architecture of the UI process is shown in Figure 5-3. Each box represents a frame within the process. The first line of text in a box is the frame name (e.g., Run-Summary) and the second line lists the main operations available from the frame (e.g, Create, Connect, . . ., Version). The frames listed in Figure 5-3 are described in more detail in chapter 4.

The User-dialog and WIP-log frames are generic frame types. In other words, there are many frames that are User-dialog or WIP-log frames. A user-dialog called from BPFL specifies the name of the User-dialog frame to call. For example, the measure-bulk-

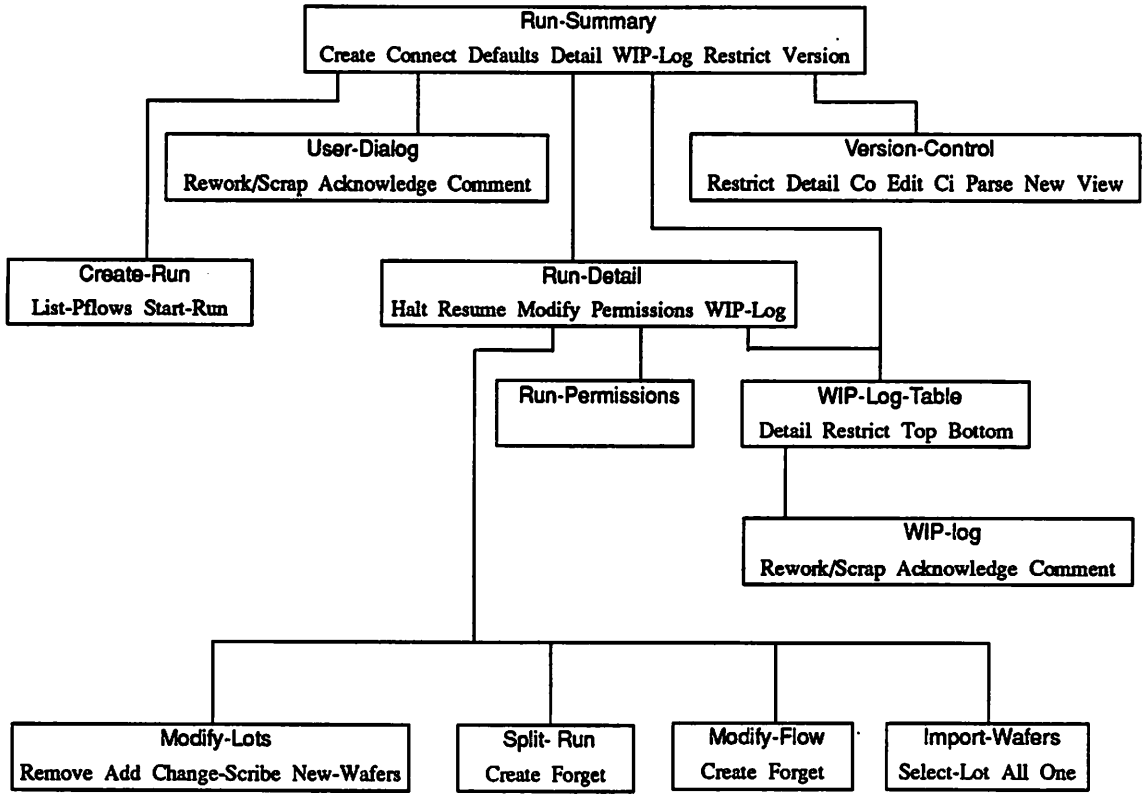

Figure 5-3: UI process application structure.

`resistivity` procedure shown in Figure 4-11 calls `user-dialog` to display the `Sonogage` frame. The `Sonogage` frame is shown in Figure 4-12. The `Sonogage-Log` frame shown in Figure 4-19 is a `WIP-log` frame which displays the log entry written by the `WIP-log` procedure call at the end of the `measure-bulk-resistivity` procedure. `User-dialog` and `WIP-log` frames are described in more detail later in this section.

The main data structures used by the UI process are the `run`, `user_dialog` and `WIP_log` tables in the database. The `run` table contains records that indicate the status of all active runs. A simplified table definition for the `run` table and some examples are shown in Table 5-1.[1] The notation used in the table is as follows. The first cell in each column has two lines of text. The

run_id (integer)	status (enumerated)	step (string)	step_path (string)	log_id (integer)	lock (enumerated)
2	running	INIT-OX	START/INIT-OX	2	free
5	waiting	PATTERN	NWELL/PATTERN	12	busy

Table 5-1: Run table definition and examples.

[1] The `run` table has many fields as shown in Appendix D.

first line is the column name, and the second line is the column type. The remaining rows give examples of values stored in the table. For example, the name of the first field is `run_id` and it contains an integer. The `run_id` field is an integer that uniquely identifies a run. The `status` field contains the current run status (e.g., `running`, `waiting`, `finished`, etc.). The `step` field contains the name of the current step and the `step_path` contains the current step-path. Both of these fields are displayed to the user in the `Run-Detail` frame. The `log_id` field identifies the sequence number of the current WIP-log entry. It is incremented each time a new log record is appended to the WIP-log. The `lock` field is used to prevent simultaneous access to a run by multiple UIPs. Lock is an enumerated field with two allowable values: `free` and `busy`. The deadlock prevention mechanism of the database management system is used to ensure mutual exclusion.

The `user_dialog` table definition is shown in Table 5-2. The `run_id` and `id` fields uniquely identify each entry in the table. The `name` string contains the name of the user-dialog frame. The `step_path` contains the run step-path at the time the dialog was saved to the database. The `procedure` field contains the name of the BPFL procedure that called `user-dialog`. The `tag` field contains the tag string supplied in the BPFL code, if any. These fields are displayed in the `user-dialog` frame. The `arguments` field contains the arguments for the frame. For example, the `measure-bulk-resistivity` procedure passes the arguments `nominal`, `limits`, and `wafer-id` to `user-dialog`. These arguments are stored in the `arguments` field in the `dialog` table.[1]

When the WIP interpreter process evaluates a `user-dialog` call, it writes the details about the dialog into the `user_dialog` table as described above. The example in Table 5-2 illus-

run_id (integer)	id (integer)	name (string)	step_path (string)	procedure (string)
2	2	Sonogage	START	measure-bulk-resistivity

tag (string)	arguments (string)
initial	:wafer-id 2 :nominal [{18 ohm-cm} {22 ohm-cm}] ...

Table 5-2: `User_dialog` table definition and example.

[1] The `user-dialog` call in Figure 4-11 also has a `tag` argument, which is extracted and passed in the `tag` field of the `dialog` table.

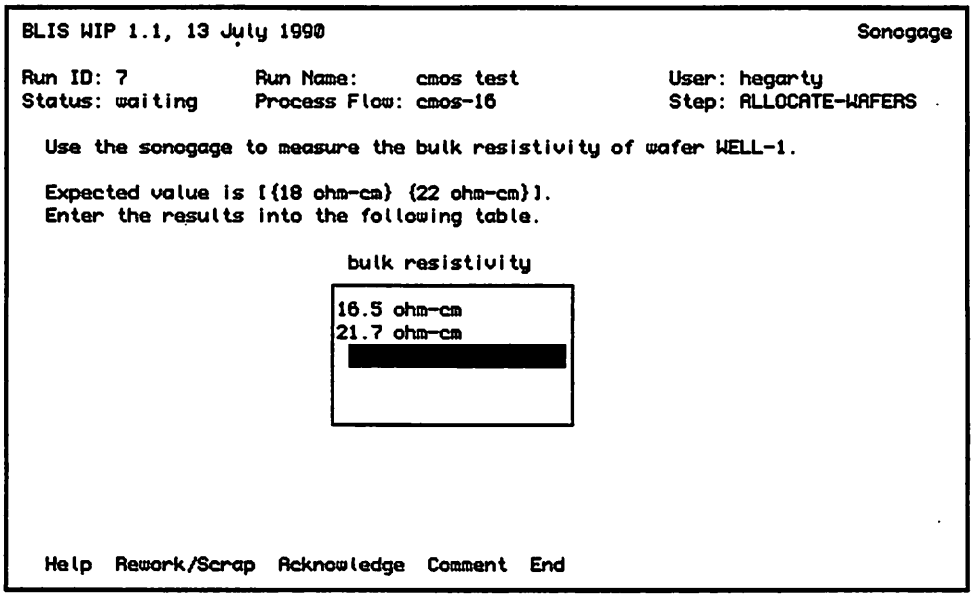

Figure 5-4: Sonogage frame.

trates what might be written for the `user-dialog` call in `measure-bulk-resistivity` when called from the process flow in Figure 4-5. When a user connects to a run, the `dialog` table is queried to select the current user-dialog for the run. The `name` field identifies the appropriate frame to display. The values in the `procedure` and `tag` fields are placed into the appropriate fields on the frame. The `arguments` field is parsed by the UI process so that the frame code can access arguments by name. Finally, the frame in the `name` field is called by the UI process.

User-dialog frames are written in INGRES *Applications By Forms* (ABF). Further detail about ABF may be found in the *INGRES ABF/4GL Reference Manual* [50]. A frame consists of two components: a *form* that displays information to the user and in which the user enters information, and a *menu* listing the available operations. All user-dialog frames support at least the following operations: `Help`, `Rework/Scrap`, `Acknowledge`, and `Comment`. A frame is defined by specifying the form layout using the forms editor and the operation is coded in ABF 4GL.[1] The `Sonogage` frame is shown in Figure 5-4. The ABF code for the operations in the frame is shown in Figure 5-5. The code is composed of a sequence of blocks that specify frame initialization and code for each operation. In the following discussion, the code in Figure 5-5 will be referred to by the line numbers on the left-hand side of the figure.

[1] 4GL is shorthand for "Fourth Generation Language."

```
1 /*
2  * Sonogage frame operation code
3 */

4 /* Frame initialization */
5 initialize (changed = integer, temp=text(80), lowr = text(20), highr = text(20),
6     x = integer, measurements.filled = integer, units = text(20)) = {
7   units := 'ohm-cm';
8   wafer_scribe := wafer_scribe(arg_pointer('wafer-id'));
9   /* Determines nominal value for measurements */
10  if arg_supplied('nominal') != 0 then
11    nominal := make_string('%s.',arg_value('nominal'));
12  else
13    set_forms field ' ' (invisible(nominal)=1);
14  endif;
15 /* Establish limits, if any */
16  if arg_supplied('limits') != 0 then
17    x := arg_pointer('limits');
18    lowr := object_printrep(interval_left(x));
19    highr := object_printrep(interval_right(x));
20  else
21    lowr := null;
22    highr := null;
23  endif;
24 /* Prepare the measurements table for the user to enter values */
25  inittable measurements fill;
26 };
27 /* Help operation definition */
28 'Help' = {
29  help_forms(subject = 'Sonogage frame', file = 'sonogage.help');
30 };
31 /* Rework/Scrap operation definition */
32 'Rework/Scrap' = {
33  callproc handle_rework_scrap;
34 };
35 /* Acknowledge operation definition */
36 'Acknowledge' (activate = 1) = {
37  callproc start_log;
38  callproc create_log_attr('measurements');
39  unloadtable measurements
40  {
41        /* append_attr_value: first arg is name of argument, second is
42           value to append, third is the type of value */
43    callproc append_attr_value('measurements',measurements.value,'{}');
44  };
45  callproc finish_log;
46  return;
47 };
48 /* Comment operation definition */
49 'Comment' (activate = 1) = {
50  callframe comment;
51 };
52 /* End operation definition */
53 'End' = {
54  return;
55 };
```

Figure 5-5: Sonogage frame function outline.

The **initialize** block initializes the values in the form. Line 7 assigns the string "ohm-cm" to the units variable.[1] Line 8 is used to display the name of the wafer (e.g., WELL-1) as shown in Figure 5-4. The arg_value function returns the value of the specified argument, in this case wafer-id. The wafer_scribe function takes a wafer-id and returns the scribe for that wafer. Lines 10–14 are used to fill in the nominal field on the form to indicate to the user the expected value of the measurement. If a nominal argument is supplied, the arg_supplied function returns a non-zero value, and the argument value is used to establish the value of the nominal field. If no nominal argument is supplied, the **set_forms** statement is used to make the nominal field invisible.

The **if** statement in line 16 is used to determine if a limits argument is supplied. The limits argument is an interval argument that specifies upper and lower limits for the values entered by the user. These upper and lower limits are stored in the lowr and highr variables respectively. If the frame has a limits argument, the code in lines 17–19 is used to store the limits in the variables. Otherwise, these variables are given null values in lines 21–22 to indicate that range checking is disabled. Line 25 prepares the measurements table field for the user to enter values. Table fields are ABF fields that display several rows and columns of data at the same time. The Sonogage frame has one table field, called measurements, into which the user enters measurements. It appears in Figure 5-4 as the box in the middle of the frame. It has a single column called value, which may be accessed by the expression measurements.value.

The code for each operation is specified in an *activation block*. Activations can be of several types. For example, lines 28–30 define an operation named Help which executes the help_forms function if the Help menu item is selected. Selecting a menu item is an example of a *menu activation*. Similarly, the Rework/Scrap operation defined in lines 32–34 calls the handle_rework_scrap function that puts up a frame suitable for reworking or scrapping wafers.

Lines 36–47 define the Acknowledge operation. Acknowledge writes a WIP-log record for this user-dialog. The start_log function is used to prepare the UI process to write out values to the log. Line 38 declares that an argument named measurements is to be written to the log. The **unloadtable** statement in lines 39–44 loops through all of the rows in the measurements

[1] ABF uses single quotes (') to delimit strings. For consistency with usage elsewhere in this dissertation, double quotes (") are used within the body of the text.

table field (i.e., where the user types the measured values) and adds the entered measurements to the `measurements` argument. The `finish_log` function saves the log to the database and sends a message to the WIP interpreter process that the run is ready to proceed. The **`return`** statement is an ABF operation which returns the UI process to the frame that called the `Sonogage` frame.

By default, all arguments supplied to the user-dialog are written to the log. Arguments can be removed or their values may be changed by the operation code if desired. A list of ABF functions supplied for writing user-dialog frames is given in Appendix C.

The `WIP-log` frame that corresponds to the `User-dialog` frame (i.e., `Sonogage-Log`) is similar in many ways to the `Sonogage` frame. Wherever possible, ABF operation code and forms are shared between frames.

Logs are stored in the `WIP_log` table shown in Table 5-3. This table is similar to the `user_dialog` table. The only difference is the addition of two fields: `user_id` and `time`. `User_id` identifies the person who entered the log record and `time` identifies the date and time at which the log record was written. The `arguments` field contains field names and values so that CLOS objects for the WIP log objects illustrated in Figure 3-5 can be created.

The UI process can perform validity checks on values entered by a user. Measurements entered into the `Sonogage` frame are checked by the code in Figure 5-6. This code defines an ABF *field activation* that is executed whenever the type-in cursor is moved out of the `value` column in the `measurements` table field. The **`inquire_forms`** statement in line 3 is used to find out if the user has changed the value in the current row of the measurements table. If not, the **`resume`** `next` operation returns control to the user after moving the type-in cursor to the next field in the form. If the value has been changed, the code in lines 9–16 is used to call the `check_format` function to parse and check the entered value.

run_id (integer)	id (integer)	user_id (integer)	name (string)	step_path (string)	procedure (string)
2	2	42	Sonogage	START	measure-bulk-resistivity

tag (string)	time	arguments (string)
initial	2/1/91 9:13	:wafer-id 2 :measurements ...

Table 5-3: `WIP_log` table definition and example.

```
 1 field measurements.value =
 2 {
 3   inquire_forms row sonogage measurements (changed = change(value));
 4   /* changed will be zero if the field has not been edited */
 5   if changed = 0 then
 6     resume next;
 7   else
 8     /* The value in the field has been changed */
 9     if lowr is null then
10       /* Recall that initialize block set lowr to null if no limit arg was passed
11          No limits argument passed - check dimensions but not range */
12        x := check_format(measurements.value,'{float}',units);
13     else
14       /* Limits argument supplied - check diemnsions and range */
15       x := check_format(measurements.value,'{float}',lowr,highr);
16     endif;
17     if x = 0 then
18       resume next;
19     else
20       resume;
21     endif;
22   endif;
23 }
```

Figure 5-6: `Measurements` table activation code.

The first argument to `check_format` is the value to be parsed. The second argument is a format string that indicates what type of value is expected. Examples include `"[integer]"`, `"complex"`, and `"[{float}]"` which specify an integer range value (e.g., `[1, 10]`), a complex number (e.g., `(1, 0.5)`[1]) and a range value of units with floating point type (e.g., `[{1.2 um}, {1.4 um}]`), respectively. Depending on the format string, `check_format` may take additional arguments. For example, if the value to be checked is a unit, the third argument is a string that contains either a dimension (e.g., `"ohm-cm"`) or a unit value (e.g., `"{12.0 ohm-cm}"`). If the third argument is a dimension, `check_format` checks the dimensionality of its second argument. If the third argument is a unit value, `check_format` takes the third argument as a lower limit of the acceptable value, and takes a fourth argument that is an upper limit.

In the `check_format` call in line 12 the second argument is `"{float},"` which specifies a unit with a floating-point number. The third argument is the string `"ohm-cm"` stored in the `units` variable in the **`initialize`** `block`. This function call parses the input value and accepts a unit value with a floating point number and a unit designator dimensionally equivalent to `ohm-cm`.[2] Standard numerical contagion rules are applied to numbers. For example, if `float` is speci-

[1] The complex number $\sigma + j\omega$ is represented by (σ, ω).

fied, both integer and floating-point values are acceptable, but complex values are not. The user need not type the square brackets for intervals or the set brackets for units as they are automatically inserted in the correct position in the parsed string. The values `"{18.2 ohm-cm}"`, `"{300 Mohm*fm}"` and `"12 nohm-parsec"` are all acceptable. Examples of unacceptable values are `"18.2"`, `"[1.2, 2.3],"` and `"{(12, -4) ohm}"`. The `check_format` call in line 15 has a similar function except that the entered value is also range-checked.

If the value entered by the user meets the required criteria, `check_format` returns zero. If the entered value is unacceptable, an appropriate error message is displayed to the user (e.g., Figure 4-13) and a negative value is returned. The code in line 18 moves the cursor on to the next field if no error was encountered while parsing the field. Otherwise, the code in line 20 leaves the cursor on the field, giving the user an opportunity to correct the problem.

5.4 Translating BPFL to Lisp

This section describes the method by which the WIP interpreter process translates BPFL code to Lisp.

Recall that early versions of BPFL used a Lisp syntax which the intended users (i.e., process engineers) found unsatisfactory. Since the current version of BPFL is block-structured, it must be translated to Lisp code for use with the WIP interpreter process. For example, the Lisp code for the `measure-bulk-resistivity` procedure in Figure 5-7 is shown in Figure 5-8. By comparing the code in Figure 5-7 with the code in Figure 5-8 it is possible to see how the translation works. The **`defflow`** construct:

```
defflow measure-bulk-resistivity (...)
let ...
begin
  ...
end;
```

is translated into the Lisp expression

```
(defflow measure-bulk-resistivity (...) (let* ...))
```

The first procedure call in the `measure-bulk-resistivity` procedure is

```
pick-test-wafer();
```

which is translated into

---

[2] Any value with the same dimensions is acceptable (e.g., `"ohm-km"`, `"(V*m^2)/(A*m)"`, etc.)

```
defflow measure-bulk-resistivity(tag)
let wafer := pick-test-wafer();
    ss := wafer-snapshot(wafer);
    seg := first(find-segments(ss, material: #m(substrate));
    mat := pif-attr-val(seg, :material, ss);
    nominal := material-attr(mat, :resistivity);
    limits := nil;
    results := nil;
begin
  if interval-p(nominal) then
    limits := make-interval(interval-min(nominal) * 0.5,
                            interval-max(nominal) * 2.0)
  else
    limits := make-interval(nominal * 0.5, nominal * 2.0);
  end;
  results := user-dialog('sonogage, tag: tag, nominal: nominal,
                         limits: limits, wafer-id: wafer-id(wafer));
  getf(results, :average) := sigfigs(average(getf(results, :measurements)),3);
  wip-log('sonogage, results);
  getf(results,:average);
end;
```

Figure 5-7: `Measure-bulk-resistivity` definition.

```
        (pick-test-wafer);
```

In general, Lisp procedure calls differ from block-structured procedure calls as follows. The block-structured call `func(b,c,d)` is equivalent to the Lisp call `(func b c d)`. The procedure argument list in the block-structured code is very similar to the argument list in the Lisp code, the only difference being that the commas between the arguments in the block-structured code have been replaced with spaces in the Lisp code. This change is generally true of all lists: block-struc-

```
(defflow measure-bulk-resistivity (&key tag)
 (let* ((wafer (pick-test-wafer))
        (ss (wafer-snapshot wafer))
        (seg (first (find-segments ss :material #m(substrate))))
        (mat (pif-attr-val seg :material ss))
        (nominal (material-attr mat :resistivity))
        (limits nil)
        (result nil))

 (if (interval-p nominal)
     (setf limits (make-interval (* (interval-min nominal) 0.5)
                                 (* (interval-max nominal) 2.0)))
   (setf limits (make-interval (* nominal 0.5) (* nominal 2.0))))
 (setf results (user-dialog 'sonogage
                            :nominal nominal :limits limits :tag tag
                            :wafer-scribe (wafer-scribe wafer)))
 (setf (getf results :average)
   (sigfigs (average (getf results :measurements)) 3))
 (wip-log 'sonogage results)
 (getf results :average)
 )
```

Figure 5-8: BPFL Lisp representation for `measure-bulk-resistivity`.

tured lists are comma delimited, whereas Lisp lists are space delimited.

All operations in Lisp are expressed as functions, so operators (e.g., assignment (: =), addition (+), etc.) are replaced by function calls.[1] For example, the block-structured statement

```
a := b + c;
```

is translated to

```
(setf a (+ b c)))
```

The semicolons that delimit statements in the block-structured code are not required in Lisp.

The only other major difference between the block-structured code in Figure 5-7 and the Lisp code in Figure 5-8 is that keywords are used differently. Keywords have two uses in BPFL. First, they specify argument names (e.g., the `measure-bulk-resistivity` procedure has an argument named `tag`). A `measure-bulk-resistivity` procedure call in the block-structured code takes the form

```
measure-bulk-resistivity(tag: "initial");
```

The corresponding Lisp function call is

```
(measure-bulk-resisitivity :tag "initial")
```

Argument names in block-structured code have the colon placed after the name, but in Lisp the colon is placed before the name.

The other use of keywords in BPFL is as attribute names. For example, the procedure call

```
material-attr(mat, :resistivity);
```

uses the keyword `:resistivity` to access the `resistivity` attribute of the material object stored in the variable `mat`. The colon is placed in front of the name in the block-structured code to avoid confusion between the two uses of keywords in Lisp.

5.5 Executing BPFL code

This section describes the method by which the WIP interpreter process executes BPFL code. The WIP interpreter process is based on the core BPFL interpreter written by Williams [37]. The core BPFL interpreter evaluates Lisp expressions and in so doing executes BPFL code.

---

[1] By convention, the Common Lisp term for *procedure* is *function*. In this dissertation, the term procedure is always used to describe BPFL code.

Slot name	Description	Example
action	Pointer to the next function to call to continue evaluation.	#<function go-funcall>
code	BPFL lisp code being evaluated.	(measure-bulk-resisitivity :tag "initial")
cp	Index into code indicating current position of evaluation.	(1)
returned-values	Slot to hold values returned by evaluation when complete.	(22.3 ohm-cm)
parent	Pointer to next frame in the stack.	#<eval-frame :id 25>
id	Integer uniquely identifying the frame.	26

Table 5-4: Simplified evaluation frame definition and example.

The evaluation of Lisp expressions yields a value. Sometimes the evaluation result is the same as the expression (e.g., the constant 5 evaluates to 5). In Lisp, this is true of numbers, strings (e.g., "abc") and keywords. Lisp keywords are symbols whose first character is a colon (e.g., :resistivity). Symbols are considered to be the names of variables and evaluate to the values stored in the variable. Examples of symbols are tag and thickness. Lists are treated as function calls and they evaluate to whatever the function call returns. Examples of lists are (setf x 5) and (a b (c d)).

Lisp code is evaluated as follows. Each time the core interpreter evaluates an expression, it creates a CLOS object called a *frame* to control the evaluation.[1] The frame is discarded when the evaluation of the expression is complete. For example, when the core interpreter evaluates the procedure call

```
(measure-bulk-resisitivity :tag "initial")
```

a frame is created to evaluate the code. When the code within the procedure is evaluated, other frames are created. The arguments to a procedure call (e.g., :tag "initial") must also be evaluated, which creates additional frames. Consequently, the interpreter maintains a *stack* of frames. The top of the stack is the frame evaluating the current expression. This frame is called the *current frame*.

A simplified definition of a frame is shown in Table 5-4. Each frame is a CLOS object that has named slots, each of which hold a value. The main slots for controlling the evaluation of BPFL

[1] Evaluation frames should not be confused with the INGRES ABF frames used by the UI process. Evaluation frames are the same as stack frames or activation records in a conventional programming language. ABF frames denote an interface abstraction.

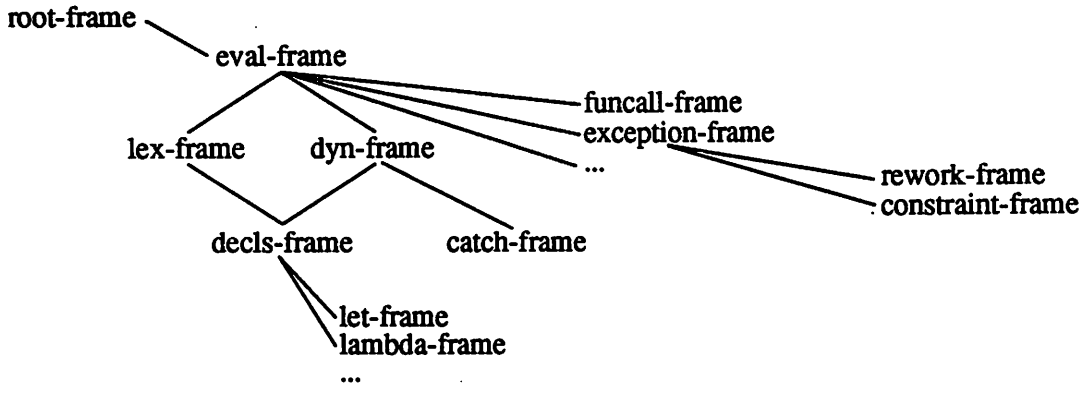

Figure 5-9: Evaluation frame class hierarchy.

code are: `action`, `code`, `cp`, `returned-values` and `parent`. The `action` slot contains a pointer to a function that is to be called to continue evaluation (e.g., `#<function go-fun-call>`).[1] The `code` slot contains the BPFL code that the frame is evaluating. `Cp` points to a position within the code which indicates the current item in the code being evaluated. The value stored in `cp` is a list indicating how many items should be skipped to read the one of interest. For example, in the list `(a b (c d))`, the symbol `a` is indexed by `(0)`, and the symbol `d` is indexed by `(2 1)`. In Table 5-4, `cp` has the value `(1)`, so the item being evaluated is the keyword `:tag`. The `returned-values` slot is used to hold values returned by the frame when evaluation is complete. For example, the `measure-bulk-resistivity` procedure returns the average of the measurements entered by the user. The `parent` slot contains a pointer to the frame whose evaluation led to the creation of this frame. The `id` slot holds an integer that uniquely identifies the frame.

Most frames contain more slots than indicated in Table 5-4. As mentioned earlier, frames are CLOS objects and the different frame types form a hierarchy as shown in Figure 5-9. The class at the base of the frame hierarchy is `root-frame`. A `root-frame` is always the first frame created when evaluating a BPFL process flow. It serves as the anchor point for the stack of frames created as the code is evaluated. Frames further down the hierarchy have additional slots. For example, `eval-frames` are used to evaluate procedural code. They have a `current-lot` slot that holds the BPFL lot `current`. Special BPFL constructs such as **`rework-loop`** and **`constraint`** require special handling during processing and frame types are defined for them.

[1] The construct `#<expr>` denotes a pointer to an `expr` object.

The process of evaluation will now be described. A `root-frame` is created with appropriate values stored in the `code` and `action` slots. The interpreter then calls the function stored in the `action` slot of the `root-frame`. This call creates a frame to evaluate the code. The interpreter repeats this process until the only frame remaining is the original `root-frame`, at which point evaluation is complete and the `returned-values` slot of the `root-frame` contains the final result.

A brief example of evaluation will be presented to illustrate how the evaluation process operates. Consider evaluation of the following code:

```
(if implant-split
  (split-lot 'product :into '(high low med))
```

The `if` statement will cause the `split-lot` function call to be interpreted if the `implant-split` variable has a non-nil value. Assume for this example that `implant-split` has the value `t` (i.e., `true`). When the interpreter begins to evaluate the `if` statement, an `if-frame` is created which becomes the topmost frame of the evaluation stack, as shown in Figure 5-10 (a). The initial value of `cp` is `(1)`, so evaluation begins with the `implant-split` symbol. The `returned-values` slot is *unbound*, meaning that it contains no value. The `action` slot points to the `go-if` function.

The frame is evaluated by calling the function pointed to by the `action` slot (i.e., `go-if`). `Go-if` evaluates the `implant-split` variable, places the value in the `returned-values` slot, and sets the `action` slot to point to the function `go-if2`. The frame then appears as in Figure 5-10 (b).

The frame is evaluated once more by calling the function pointed to by the `action` slot. The action in this case is to examine the value stored in the `returned-values` slot and, if it is non-nil, as in this example, execute the code within the `if` statement (i.e., `(split-lot ...)`). Since the code is a function call, the evaluation creates a `funcall-frame` as shown in Figure 5-10 (c). This frame becomes the current-frame for the evaluation. Note that the `parent` slot of the `funcall-frame` points to the parent of the `if-frame`. This is because after the evaluation of `split-lot`, the `if` frame is no longer required, and the parent-frame of the next frame created after the `if-frame` is the parent of the `if-frame`. Consequently, the `if-frame` is discarded.

```
if-frame
          action:   #<function go-if>
            code:   (if implant-split ...)
              cp:   (1)
 returned-values:   unbound
              id:   25
          parent:   #<eval-frame :id 24>
```

(a)

```
if-frame
          action:   #<function go-if2>
            code:   (if implant-split ...)
              cp:   (2)
 returned-values:   true
              id:   25
          parent:   #<eval-frame :id 24>
```

(b)

```
funcall-frame
          action:   #<function go-funcall>
            code:   (split-lot 'product :into '(low med high)
              cp:   (1)
 returned-values:   unbound
              id:   26
          parent:   #<eval-frame :id 24>
```

(c)

Figure 5-10: Evaluation examples.

This example is somewhat simplified. More detail about evaluation is presented by Williams [37].

5.6 Saving Run State

This section describes the means by which the WIP interpreter process saves the state of a run in the database. Run state is saved whenever a run is suspended (e.g., when waiting for a user or piece of equipment to complete an operation). Run state is saved for two reasons. First, it provides a high degree of software fault-tolerance so that run execution will not be disturbed if the computer system crashes. Second, it is easy to modify a run that has been saved in the database.

The core interpreter described in section 5.5 is well suited for use by the WIP interpreter process because the iterative evaluation process provides natural points at which to save the state of the run. A simplified definition of the `run` structure is shown in Table 5-5. The complete data structure definition and all subsidiary structures are described in Appendix E. The `id` slot uniquely

Slot name	Description
id	Integer uniquely identifying the run
current-frame	Frame at the top of evaluation stack.
root-frame	Frame at the base of the evaluation stack.
bindings	Global variable bindings used in the BPFL code.
wafer-lot-state	Pointer to object describing the wafers and lots in the run.
materials	Pointer to object describing the run materials.
layers	Pointer to object describing run layers.
masks	Pointer to object describing run masks.
snapshots	Pointer to object describing snapshots used by wafers in the run.
exception-frames	List of exception frames for exception handling.
module-id-list	List of id's of BPFL code modules used by the run.

Table 5-5: Run data structure definition.

identifies the run. The current-frame and root-frame slots were described above. The bindings slot holds global variable and constant bindings (i.e., variables or constants whose value is accessible from all BPFL code in the run). Examples of global constants are *mask-set* and *product-lot-size*.[1] The wafer-lot-state, materials, layers, masks, and snapshots slots all contain pointers to data structures that describe entities manipulated by the run (e.g., materials, masks, etc.) and computation entities (e.g., conditions). Exception-frames list frames that correspond to each exception activation. The implementation of exceptions is described later in this chapter. The module-id-list contains a list of the code modules used by the run. It is used to update process-flow code.

Recall that CLOS instances are written to the database by methods on each class. Consider, for example, the methods for saving evaluation frames. Frames are written into the evaluation_frame table defined in Table 5-6. The run_id field contains the integer that identifies the run to which the frame belongs. Similarly, the frame_id field contains the integer that identifies the frame within the run. The frame_type field contains the name of the frame class. The remaining slots in the CLOS instance are written into the frame_slots field. Because the width of the

run_id (integer)	frame_id (integer)	frame_type (string)	frame_slots (string)	extend (integer)
1	25	if-frame	:action #'go-if :cp (1) : ...	0
1	24	funcall-frame	:action #'go-funcall :cp (1) ...	0

Table 5-6: Evaluation_frame table definition and examples.

[1] BPFL has adopted the Lisp convention of using asterisks around the names of global variables.

Slot name	Description	Example
snapshot	Pointer to PIF snapshot describing the wafer state.	#<Snapshot :id 3>
id	Integer uniquely identifying the wafer within the fab.	4002
index	Integer uniquely identifying the wafer within the run.	27
scribe	String containing the wafer scribe.	"NWELL-1"

Table 5-7: `Wafer` class definition and example.

`frame_slots` field is limited by the database, it is possible that the field may not be wide enough to hold all of the slots for a given frame.[1] For this reason, an `extend` field is used to indicate when a frame definition must occupy multiple rows in the table. The value of `extend` is an integer, starting at 0 for the first row describing a given frame, and incremented for each subsequent row.

Frames are written to the database using a method named `db-print-frame`. The method takes a frame object as an argument and returns a string that contains the frame slots written out in a format that can be read back correctly. For example, if the `cp` slot has the value `(1 2)`, the slot name and value are written as `":cp (1 2)."` The `action` slot in an evaluation frame contains a pointer to a function. The name of the function is written out instead of the hexadecimal memory address stored in the pointer because the frame may need to be read into a different version of the WIP interpreter process which might store the function at a different location in memory. For example, the function `#<function go-if>` is written out as `":action #'go-if"`. The characters `#'` indicate a Lisp *reader macro* [39]. Reader macros are used to change the interpretation of the item that follows them (e.g., the symbol `go-if`). The reader macro `#'` returns a pointer to the function whose name is the symbol following it. In this case, when the string `"#'go-if"` is read, a pointer to the function `go-if` is returned.

The same mechanism is used for other slots that are pointers. For example, consider the `parent` slot. The value stored in this slot is a pointer to an evaluation frame, so the storage method represents the frame by the value of the `id` slot of the frame pointed to by `parent`. The `code` slot is treated specially because of the requirement to change code while a process is running. Dynamic code modification is discussed in section 5.9.

Frames are read into a WIP interpreter process by creating an instance of the CLOS class stored in the `frame_type` slot and passing the contents of the `frame_slots` field to an `ini-`

[1] The current version of INGRES limits a record to 2 K bytes.

run_id (integer)	id (integer)	index (integer)	snapshot_id (integer)	scribe (string)
1	4002	27	3	NWELL-1

Table 5-8: `Wafer` table definition and example.

`tialize-instance` method for that class. `Initialize_instance` sets up the values in the slots.

Some run data structures in the database must be accessible from the UI process (e.g., wafer and lot information). Since the UI process is an ABF application with low-level routines written in C [55], it is important that the data structures which it must read are easy to access from C. For example, BPFL represents wafers and lots as CLOS objects. The slots for the `wafer` class are shown in Table 5-7. The database table used to represent this class is the `wafers` table defined in Table 5-8. The `wafer` table can be read by the UI process because each slot is stored in a separate field, and the value in each field is easy to interpret (i.e., an integer or string).

The slots in the `lot` class are shown in Table 5-9. The `bits` slot is an integer that indicates which wafers are present in the lot. Recall that each of the n wafers allocated to a run has a unique `index` in the range $[1, n]$. The least-significant bit in `bits` represents the wafer with an `index` of 1. The next least-significant bit represent the wafer with an `index` of 2 and so forth. For example, the value of `bits` in the example in Table 5-9 is binary `101001110`, so the lot in the example contains the wafers 2, 3, 4, 7, and 9.

This approach to representing the wafers present in a lot is very compact and has much to recommend it. Because Common Lisp supports integers of arbitrary length, there is no danger of the number of wafers exceeding the number of bits that can be represented in an integer. However, neither INGRES nor ABF supports arbitrarily wide integers. Consequently, a different representation for lots is needed in the database.

The database representation for `lots` is shown in Table 5-10. The `run_id` and `id` fields uniquely identify the lot. The `name` field is a string that contains the name of the lot.[1] `Bits` is a 32-

Slot name	Description	Example
`id`	Integer uniquely identifying the lot.	2
`bits`	Integer representing wafers present in the lot.	334

Table 5-9: `Lot` class definition and example.

run_id (integer)	id (integer)	name (string)	bits (integer)	lsb (integer)
1	1	SPLIT-LOW	334	1

Table 5-10: `Lot` table definition and example.

bit wide integer field that contains 32 bits of the `bits` slot in the `lot` class. If the `bits` slot is wider than 32 bits, it is split across multiple rows in the database. The `lsb` field in the table specifies the `index` corresponding to the wafer represented by the least-significant bit in the `bits` field.

This approach provides a way to store the arbitrarily long `bits` slot while making it easy for the UI process to determine which wafers belong to a lot. This information is needed to display the lot data to the user as shown in the `Run-Detail` frame in Figure 4-16.

The WIP interpreter process saves run state whenever a run is suspended. If the system crashes while a run is suspended, the run can be restored and executed with no loss of data. If the system crashes during the brief period in which the state is being saved, recovery depends upon the database. If the system crashes while a run is executing, no data is lost, but when the run is restored it must begin executing code from the point at which it was last saved. For this reason, run state is always saved *in toto*. It would be more efficient to save parts of the run state to the database (e.g., the wafer state) immediately any changes are made to the data structures representing them. However, this approach makes it impossible to guarantee that execution can always be carried forward from the last saved state since different parts of the run state have been saved at different times.

5.7 WIP interpreter operation

This section describes the operation of the WIP interpreter process and its interaction with other components of the WIP system.

The WIP interpreter process is a server process that executes several runs concurrently. The architecture of the system is based on an *event-loop* which processes events as they are received. Events are generated by the UI process, the EI process and the WIP interpreter process itself. The main loop is shown in Figure 5-11. The first action in the loop is to process any pending events. The event types and their function are:

1. `Start-run` – create and begin execution of a run. A `start-run` event is

[1] The `lot` class definition in Table 5-9 does not contain a `name` slot. The WIP interpreter process maps lot names to the corresponding `lot` class via a hash-table.

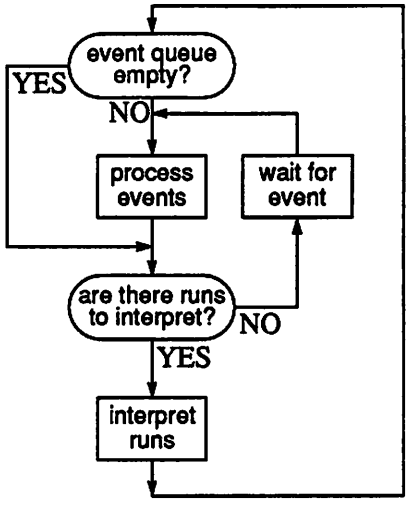

Figure 5-11: WIP interpreter process main loop.

created by the UI process when the user selects the `Start-run` operation in the `Create-run` frame shown in Figure 4-4.

2. `Suspend-run` event – stop execution of a run. For example, the UI process generates a `suspend-run` event if the user selects the `Halt` operation in the `Run-detail` frame in Figure 4-16. Similarly, the WIP interpreter process itself generates a `suspend-run` event when a `user-dialog` is executed.
3. `Restart-run` – resume execution of a run.
4. `Run-complete` – mark run as finished.
5. `Message` – send message to users or equipment. For example, a message is sent to a user when an error occurs during run processing. The message is either sent to the controlling UI process for the run, or if no such UI process exists, electronic mail is sent to the run owner. Equipment messages are sent to the appropriate EI process for the specified piece of equipment.
6. `Exception` – raise exception. These events are used for rework, constraint, and general exception handling as will be described in the next section.
7. `Shutdown` – terminate the WIP interpreter process. This event completes current operations, saves the state of all active runs to the database, sends messages to all active users indicating that the WIP interpreter process is terminating,

and exits. Such an event is necessary to ensure that all state information is saved before the WIP interpreter process is shut down.

8. `Module` – performs an action on a code module. These events are used to force the WIP interpreter process to perform some action on a BPFL code module. For example, the version control system may ask that every run using a particular module be updated to use a different module. Code modules are described in section 5.9.

The WIP interpreter process is written so that unauthorized or improper use of events is prohibited. For example, a user cannot suspend a run for which he or she does not have access permission. Similarly, only `shutdown` events originating from users with WIP interpreter process system privileges are executed. These security checks are carried out within the WIP interpreter process itself because the connections between the various WIP components are not secure.

Events are high-priority items that require attention. Consequently, all pending events are handled by the WIP interpreter process before any other operations are performed. This approach was chosen to provide maximum responsiveness to interactive requests. As shown in Figure 5-11, once the WIP interpreter process has processed all events in the event queue, it continues execution of active runs. Each active run is advanced by a single function evaluation like the evaluation shown in Figure 5-10. When the WIP interpreter process has no events to process and no runs to execute, it sleeps. When an event is received on one of the TCP/IP connections to other WIP system components, the WIP interpreter process resumes execution.

The current implementation of the WIP interpreter process is intended for use in a low-volume development fab such as the UC Berkeley Microlab. In this environment, most processing on a run is carried out by one user. The WIP interpreter process sends all user-dialog requests and messages to that user. As the user moves within the fab and logs on to different terminals, the WIP system sends messages to the terminal where the user is working. The WIP interpreter process uses the identity of the user processing the lots to determine where to send requests.

In a production fab, lab technicians operate equipment in workcells. Workcells receive lots for processing, perform the necessary operations on the lots, and pass them to the next workcell. In that environment, when a user-dialog or message is generated, the WIP interpreter process sends a

request to the UI process running at the workcell that manages the equipment required for the operation. In other words, the WIP interpreter process uses the location of allocated equipment that will perform the processing operation. The two modes (i.e., user-centered and equipment-centered) are not mutually exclusive, and the WIP interpreter process can be configured to choose between them as appropriate. For example, user-dialog requests could be sent to equipment locations, but status messages (e.g., run-complete) might be sent to the user in charge of the run.

The remainder of this section reports on a coarse performance evaluation of the WIP interpreter process. Six types of operations were executed to test the performance of the system:

1. create a run,
2. save run state,
3. retrieve run state,
4. interpret a simple loop,
5. interpret a simple loop with a single Lisp function call in it, and
6. interpret a simple loop with a single BPFL procedure call in it.

All tests were performed on a Sun Sparcstation 1 running SunOS 4.0.1 using Allegro Common Lisp 3.1 [56]. The database used was release 6.3 of INGRES [53] running on a local SCSI[1] disk. Tests were carried out using the `user-time` value returned by the Lisp `time` function. Each test was repeated five times and the average value of the measurements was taken.

The time required to create a run using the `cmos-16` process flow was 2.81 seconds. The time required to save run state varies with the size of the run information (e.g., the number of stack frames), but for a run using `cmos-16` with 35 stack frames, 4.53 seconds was required. Retrieving run state took 3.05 seconds. These operations occur infrequently. For example, run state is saved only when a run is suspended, which occurs approximately 150 times in the course of executing a complete `cmos-16` process flow. In the very unlikely event that rework could double the number of run suspensions, and given that the throughput time for the run varies between 2 weeks to 3 months, the WIP interpreter process can expect to spend at worst 0.2% of its time saving and restoring run state for any given run. At worst a run might be suspended and restarted 5 times in one

[1] Small Computer Systems Interface.

```
for i := 1 to 100 do
  code-body
end;
```

Figure 5-12: Interpreter test code fragment.

hour, resulting in the WIP interpreter process spending 0.6% of its time saving the state for one run.[1]

These results measure the performance of Lisp routines built into the WIP interpreter process. Most of the time required to create, save and retrieve run state is consumed by the database operations. Recall that the database is stored on a a SCSI disk. SCSI is a low-performance interface, and much faster performance is possible with appropriate hardware. The remaining tests measure the speed of BPFL code evaluation.

The code fragment used to evaluate interpretation speed is the **for** loop shown in Figure 5-12, which executes the *code-body* code 100 times. Translation of the code to Lisp and subsequent macro-expansion results in the interpreter executing the following Lisp code:

```
(let ((i 1))
  (declare (type (integer 1 101) i))
  (block nil
    (tagbody #:g2615
      (if (> i 100)
          (progn (return-from nil (progn)))
        nil)
      ; code-body
      (setq i (1+ i))
      (go #:g2615))))))
```

This code requires 13 evaluations for each execution of the loop, plus the number of evaluations required for the *code-body* code. Three different *code-bodies* were tested. The results are summarized in Table 5-11. The null *code-body* test indicates that the WIP interpreter process has an average evaluation time of 7.8 ms for each interpreter evaluation. The *code-body* in the second and third

Loop body	Execution time (seconds)	Evaluations per loop
Null	10.2	13
Lisp function call	12.1	16
BPFL procedure call	14.9	17

Table 5-11: WIP interpreter process evaluation times.

[1] The WIP interpreter process does not need to retrieve run state from the database unless it has crashed and been restarted.

tests called the function `(loop-test i)`. In the second test, `loop-test` was a Lisp function:

```
(defun loop-test (a)
  t)
```

In the third test, `loop-test` was a BPFL procedure:

```
defflow loop-test (a)
begin
  t;
end;
```

In both cases the function takes one argument and returns the value `t`. The evaluation time for the WIP interpreter process is dependent on the evaluation performed. For example, the average evaluation time calculated from the second test is 7.6 ms, which shows that the Lisp function call is relatively fast and decreases the average evaluation time. The third test has an average evaluation time of 8.8 ms, indicating that the BPFL procedure call is relatively slow.

As an example of the evaluation time on real code, the `measure-bulk-resistivity` procedure in Figure 5-7 was evaluated. The WIP interpreter process took 1.55 seconds to perform the 190 evaluations required to execute the code. The average evaluation time was 8.2 ms. Approximately 86% of the execution time was spent in creating evaluation frames, and a further 11% was spent in CLOS method calls on evaluation frames. Recoding the interpreter to use Lisp structures rather than CLOS classes for evaluation frames should result in a ten- to one-hundred fold speed increase. Recoding the interpreter in C or C++ [57] should result in a one-hundred to one-thousand fold speed increase. The code size of the Common Lisp WIP interpreter is 16 megabytes. By recoding in C++, the executable could be reduced to approximately half that size.

To illustrate the significance of the evaluation performance figures, consider the `cmos-16` process flow. Execution of the code up to the first implantation step requires 1320 evaluations, for an execution time of 10.8 seconds. This section of code has 9 user-dialogs or equipment operations, and takes approximately one day to process in the fab. The execution time of the code is relatively small compared to the 41 seconds required to save the run state for the 9 times the run is suspended. The average time spent by the WIP interpreter process saving run state or interpreting code for this example is 0.06%. This is more than fast enough for use in a research fab. Since the execution time of the process-flow code can readily be reduced by several orders of magnitude, the execution time of the code is not significant. However, increasing the speed of database access is essential, not only

for the WIP system but for all other CIM applications. A production fab requires a high-performance database server.

The WIP interpreter process is very slow compared to most program execution systems. However, it is fast enough for the tasks it is intended to perform. Complex programs (e.g., numerical analysis routines in recipe generators) are prohibitively slow using the prototype implementation of the WIP interpreter process. If BPFL process flows require such functionality in the prototype implementation, there are two approaches to providing it. First, the routines could be written in another language (e.g., C++ or Lisp) and called from BPFL process flows. The WIP interpreter process uses this approach to implement many standard BPFL procedures (e.g., `find-surface-segments`, `format`, etc.). If users adopt this approach for writing process flow code, the disadvantage is that parts of the process flow are written in a different language. An alternative approach is to use the simulation interpreter written by Williams [37]. The simulation interpreter translates BPFL code into pure Lisp which can then be compiled. For example, the simulation interpreter runs the loop test with the Lisp function call approximately 78,000 times faster than the WIP interpreter process. A large part of this speedup is due to the fact that no CLOS object creation or method calls are involved in the execution of the compiled code. The disadvantage of this approach is that the simulation interpreter does not permit code modification, saving and restoring run state, or running the interpreter as a server. However, BPFL utility routines (e.g., to calculate the parameters for an equipment recipe) could be compiled and called from the WIP interpreter process. This approach would only work for procedures that did not need to be suspended (i.e., no `user-dialog` or `run-recipe` procedure calls are permitted). Recoding the interpreter for speed will make this approach unnecessary.

5.8 Rework, Exception, and Constraint Implementation

The core interpreter is written as a *single-threaded* process. That is, when BPFL code is being evaluated, at any time the evaluation stack represents the evaluation of a single fragment of BPFL code. This situation is illustrated on the left-hand side of Figure 5-13. Boxes represent evaluation frames. The first line in a box is the frame type and its `id`. The remaining lines show the code being evaluated by the frame. Each frame has at most one parent and one child frame, indicat-

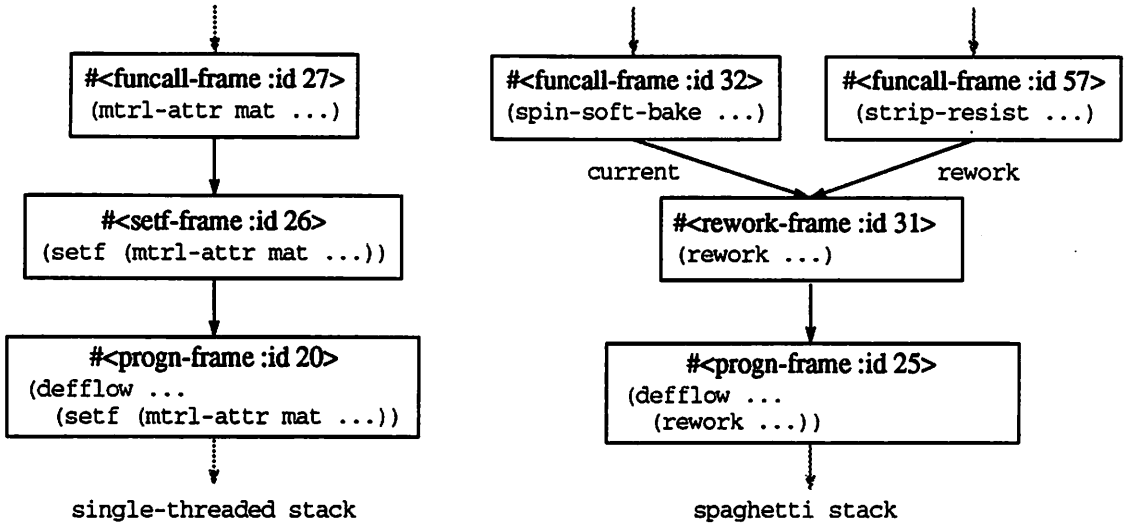

Figure 5-13: Spaghetti stack example.

ed by an arrow (the parent is at the arrowhead end of the line). For example, frame 20 is the parent of frame 26, and frame 27 is the child of frame 26.

Rework and constraints complicate this model of execution. For example, whenever a run is executing BPFL code within a **rework-loop**, the user can force rework to occur on a chosen set of wafers. The semantics of the **rework-loop** operation require that the rework lot be processed by the **rework-prefix** code, then given the same treatment as the rest of the wafers. Figure 5-14 shows an example of rework from the `pattern` procedure in Figure 3-12. Suppose that the code being executed is part of the code inside the rework body (e.g., inside the `expose-resist` procedure). At this point, the stack looks much like the single-threaded stack on the left-hand side of Figure 5-13. Suppose that the user raises a `rework` exception, which causes processing on the `current` lot to stop and processing on the `rework` lot to begin. The evaluation of the code for the `current` lot is resumed later so the evaluation stack for the `current` lot must be

```
rework-loop
  /* rework body */
  spin-soft-bake(double-photo: double-photo);
  expose-resist(mask-name: mask-name);
  develop-resist();
  rework-test inspect-resist();
  retry-count 5;
  rework-prefix if not(double-photo) then strip-resist() end;
end;
```

Figure 5-14: `Rework` example.

```
handler-case
  download-recipe(...);
  start-recipe(...);
  on-exception c := equipment-error do
    report-error("Error occurred during run-recipe: ~s", c);
    halt-run();
  end;
end;
```

Figure 5-15: Handler-case example.

preserved. The rework lot is processed by the **rework-prefix** code and then by the rework body. Evaluating the code for the rework lot results in a second stack of frames originating from the rework frame. At this point, the stack looks like the right hand side of Figure 5-13. The stack marked current is the code evaluation stack for the current lot, and the stack marked rework is the code evaluation stack for the rework lot. In other words, the rework-frame has two child frames (i.e., frames 32 and 57 are children of frame 31). This situation cannot arise with a single-threaded stack.

The semantics of rework require that rework branches be merged when the rework lot reaches the same processing point as the execution path that generated the rework. For example, when the execution state of the frame at the top of the stack above frame 57 reaches the same point as the execution state of the frame at the top of the stack above frame 32, (i.e., the point at which the rework occurred), the two paths must be merged so that processing can proceed on all of the active wafers. Given that rework loops can nest and that occasionally multiple rework will be forced from within the one rework loop (i.e., a rework can be forced while a group of reworked wafers is being processed), it is clear that the evaluation stack diagram can have multiple branch points and that a frame can have any number of children.

Rework is an example of exception handling, and the rework-frame is a special version of the exception-frame as shown in Figure 5-9. The exception-handler in the run-recipe procedure in Figure 3-7, shown in Figure 5-15, will be used as an example of exception handling. Recall that **handler-case** executes a body of code with exception handlers defined that will be called if an exception is raised (see section 3.4). If an equipment-error exception is signalled within the code in the body of the **handler-case**, the code in the equipment-error clause is executed. Several procedures are provided in BPFL that can be used to manipulate a run within an

module_id (integer)	name (string)	code (string)	extend (integer)
10	measure-bulk-resistivity	(defflow measure-bulk-resistivity (&key tag ...))	0

Table 5-14: Procedure table definition and example.

cedures in the database if they have not already been saved.

BPFL procedures are saved in the procedure table shown in Table 5-14. The module_id field is an integer that identifies to which code module the procedure belongs. The name field is a string that specifies the name of the procedure. The code field contains the Lisp code for the procedure. The extend field is used to store procedures with more code than will fit into one record.

The code slot in a frame holds the BPFL code being evaluated by that frame. This code is not saved to the database with the frame, for three reasons. First, the WIP interpreter process stores procedure code from all BPFL modules in the database separately as described in the next section. Second, storing the code with the frame is wasteful because during evaluation of a section of code, many frames will have code slots that contain fragments of the same BPFL code. Third, modification of the code used by a running process is much easier if the code is not stored with the frames.

The database representation of BPFL code stored in the code slots in frames is as follows. Frames that are created when a procedure is called are called funcall-frames. The code slot stored in the database for a funcall-frame is the name of the procedure and the id of the package to which it belongs. For example, if a funcall-frame is created to call the procedure pattern in the litho library version 1.0, Table 5-13 shows that the module_id for the module is 4. The value stored in the code slot of the frame in the database is (:procedure pattern :module-id 4). When the frames are recreated from the information stored in the database, the procedure table in Table 5-14 is used to restore the correct value in the code slot.

During evaluation of pattern, evaluation frames will be created with code slots that are fragments of the code in pattern. For these frames, the value stored in the code slot in the database is a pointer into the code for the procedure. Recall that code pointers are represented by a list indicating how many items should be skipped to read the one of interest.

5.10 Run Modification

This section describes the implementation of dynamic run modification. A user can perform modification of an active run. The allowable modifications are:

1. add or remove wafers,
2. import wafers from another run,
3. split a run into multiple runs, and
4. modify the process flow code used by the run.

Wafer and lot manipulation is specified through the `Modify-Lots` UI frame shown in Figure 4-20. Adding and removing wafers and moving them between lots involves a straightforward manipulation of the `wafer` and `lot` tables shown in Table 5-8 and Table 5-10, respectively. The only difficulty is that the WIP interpreter process must prevent the removal of a wafer that is currently bound to a variable. For example, if a BPFL run is executing the `measure-bulk-resistivity` procedure in Figure 4-11 and the `wafer` variable has a wafer assigned to it, that wafer cannot be deallocated. In general, the WIP interpreter process disallows the deletion of run structures if those structures are currently referenced by variables or arguments. If this capability is required, a symbolic debugger for BPFL code is necessary. A debugger allows variable and argument values to be changed after they have been evaluated.

Importing wafers from another run requires that the snapshots for the wafers be added to the run structure. PIF attributes for the wafers being imported normally have to be renumbered to prevent `id` conflicts between the snapshots already in the run. Creating new wafers requires the specification of a BPFL procedure call to establish the initial snapshot for the wafers, as shown in Figure 4-21.

Splitting a run into multiple runs is implemented by copying the run data structures for each of the new runs. None of the data structures are shared because each run must be free to modify the structures independently. Before a run can be split, all pending equipment operations must be complete. In addition, any pending constraints are duplicated for each of the new runs.

The user enters data into the `Modify-Flow` UI frame shown in Figure 4-24 to indicate how the WIP interpreter process should respond to code modified in modules used by a run. As explained in section 4.5, there are three possible responses to changes in the code in a module:

1. `Static` means never update the code in a module,
2. `Latest` means to always update to the latest version, and
3. `Query` means to ask the user before updating.

The prototype WIP system imposes two restrictions on code modification. First, it does not allow procedures that are currently being evaluated to be deleted. Second, it does not allow code that has already been evaluated to be modified. For example, if the `measure-bulk-resistivity` procedure in Figure 4-11 has been evaluated up to the code

```
results := user-dialog('sonogage, nominal: nominal, limits: limits,
                       :tag tag wafer-id: wafer-id(wafer));
```

the WIP interpreter process will disallow any changes to the code before this point. The reason for this restriction is two-fold. First, it is impossible in general to change the run state to make it appear as if the run had been executed on the modified code. For example, if the line of code

```
nominal := material-attr(mat, :resistivity);
```

is modified to

```
nominal := material-attr(mat, :bulk-resistivity);
```

the value of the `nominal` variable would have to be updated to reflect this change. Such changes are impossible to perform with the current system. The second reason for disallowing code changes before the evaluation point is that it makes it difficult to ensure that the `code` slots of runs have the correct code stored in them. If a user modifies code before the evaluation point, the changes in the procedure are ignored until the next time the procedure is called.

5.11 Implementation Environment

The WIP interpreter process is written in Common Lisp. The programming environment is Allegro Common Lisp using the Xerox Portable Common Loops (PCL) implementation of CLOS. Common Lisp is the implementation language for three reasons. First, it is easy to develop programs in Lisp that manipulate other Lisp programs since a program is represented by list data structures in Lisp. Second, Lisp provides a powerful and flexible framework within which to experiment with language designs. During the course of this research, many modifications were made to BPFL and Lisp greatly reduced the amount of work necessary to make these changes. Third, Lisp has a built-in evaluator that makes it easy to implement language interpreters.

Common Lisp has excellent debugging tools. Code can be compiled with extensive run-time checking to catch programming errors. Most errors generate precise messages that identify the source of the problem. Some implementations of other languages (e.g., C or C++) do not provide the same degree of run-time checking so errors tend to result in program crashes, and the source of the error must be found by invoking a debugger on the program core file. Error messages are usually much less precise about the source of the error than the Lisp messages.

The disadvantage of Common Lisp is that it is slow. Anecdotal evidence suggests that re-coding Lisp programs in C results in a 5–10 times increase in execution speed and 50% reduction in executable-code size. Much greater speed benefit can be derived in this situation because CLOS can be replaced with a faster object system. This approach is the likely choice for commercializing BPFL. Recoding the prototype in C is a relatively easy task compared to writing the system from scratch in that language. If C or C++ had been used as the implementation language, many of the changes made to BPFL would have required extensive rewriting of the interpreter.

The experience of writing the WIP interpreter underlines the need for a relational or object-oriented database and a programming language that can store persistent data types in the database. For example, the current implementation saves most slots in CLOS instances by concatenating the names and values of the slots into a large string and writing the string into a field in the database. This approach was chosen because it simplified writing the methods for storing and retrieving instances. However, database queries cannot be qualified by the values stored in those slots. Slots that are used in qualifications are saved in separate fields. This is not a significant limitation at present, but in the future databases will support more data types and many of the concatenated slots could reasonably be used in qualifications. Ideally, the database and language should permit storage of complex objects and qualifications on any slot.

The UI process is written in ABF. ABF is essentially C with high-level constructs for coding UI frames and accessing the database. ABF is a good tool for those tasks, and since manipulating frames and accessing the database are the main activities of the UI process, using ABF was much simpler than using C. Using Common Lisp was unacceptable because the UI process has to be small so that many copies of it can run at the same time.

However, ABF suffers from a number of limitations that make it difficult to use for writing large extensible programs. For example, there is no way to declare global constants or variables. Because ABF is based on C, the limitations of C make some operations (e.g., manipulating units, complex numbers, or intervals) difficult and inelegant. An object-oriented version of ABF that included richer data structure primitives would simplify the implementation of these operations.

The database used in the WIP system is INGRES, a commercial SQL DBMS. It has a powerful set of utilities for managing databases. The main limitation of INGRES for the WIP system is that it does not support structured data types (e.g., intervals, arrays). The latest version of INGRES (release 6.4) allows the user to add new data types so it is likely that support for some structured data types can be added to the WIP database. Furthermore, INGRES now automatically creates unique entity identifiers which simplifies saving and restoring run state.

5.12 Summary

This chapter has described the operation of the WIP system that is the subject of this dissertation. Communication between the processes that make up the WIP system is via TCP/IP connections and through a shared database. New `WIP-log` and `User-dialog` frames can be defined by users and added to the user-interface process. The WIP interpreter process executes BPFL code by interpreting a Lisp version of BPFL. The interpreter saves the state of runs to the database to provide software fault-tolerance and permit run modification.

[This page intentionally blank]

Chapter 6
Conclusions

The problem this thesis attempts to solve is the development of a WIP system using a powerful SPR to overcome some of the shortcomings of existing systems. Some of these shortcomings include multiple representations of process flows, limited control flow and exception handling, and flexible interaction with the other components of a CIM system. This chapter discusses the major contributions of the work and suggests future research.

6.1 Major contributions

The major contributions of this work are the development of a fault-tolerant WIP system and the design and implementation of a run management system that allows dynamic modification of runs.

Much of the power in the WIP system is derived from the capabilities of BPFL. The core features of the language, including the language notation, materials, unit and interval data types, and the wafer-state representation were developed by Williams [37]. However, many features of the language developed for fabrication in this dissertation were designed to solve problems encountered with early versions of the WIP system. These features include exception handling, constraints, rework, and some of the high-level abstractions for specifying semiconductor operations.

We believe that the exception-handling facility in BPFL is unique among other SPR languages. And, as noted several times in previous chapters, we also believe that exception handling is an essential part of an SPR because unexpected situations occur frequently in processing and there must be some high-level mechanism for dealing with them.

Similarly, constraints give BPFL a unique capability to express the semantics of fabrication operations not available in most SPRs. Constraints on processing operations, particularly timing constraints, are common in semiconductor processing. Structured documentation and run sheet systems provide some mechanism of alerting the user about constraints, but they provide no mechanism to enforce them. MIT's PFR [20] permits the specification of the time required by an operation and the permissible delay between operations, but the `constrain` construct in BPFL provides both greater flexibility and – more importantly – specifies operations to perform when a constraint

is violated. In other words, BPFL captures both the specification of the constraint and the action to take to recover from a constraint violation. Furthermore, BPFL constraints can be arbitrary, so the specification of constraints other than timing constraints are feasible (e.g., temperature or humidity limits in the fab).

The `defequipment` declaration, and the `run-recipe` and `user-dialog` procedures provide a mechanism for separating facility-specific detail from the process specification. Some SPRs make little effort to separate the facility and process specifications. We believe that a separation is required to permit processes to be moved between facilities, which remains one of the biggest problems facing the semiconductor CIM community.

The run management system has several important capabilities. First, the version control system for process flows provides a simple mechanism to track modification to flows and to control who can use and modify a process flow. Coupled with the ability of the WIP system to dynamically modify the process-flow code used by an active run, BPFL overcomes a major shortcoming of procedural SPRs, the inability to easily modify the process used by a run once it has been started.

Other dynamic modification features are the ability to move wafers between runs and split runs. The ability to move wafers between runs is of great value in a research environment. While such capability exists with structured documentation and run-sheet systems, a major advantage of BPFL is that because wafer state is transferred with the wafers, it is possible to prevent improper treatment of wafers. For example, placing wafers with Ta_2O_5 dielectric into a gate oxidation furnace will seriously contaminate the furnace. Such occurrences are common when wafers are moved between processes and no mechanism exists to capture wafer state. The proper use of the BPFL wafer-state model can prevent such accidents.

6.2 Future Research

The current version of BPFL can certainly be improved. A mechanism is needed that allows separate attributes to be associated with an operation in a procedure. For example, the operations to generate instructions to a user to perform an oxidation must occur within the same body of code as the PIF commands to update the wafer state. Ideally, the PIF operations should be generated from the manufacturing specification. Furthermore, the PIF operations should be in a separate body of code so that they will only be executed if the oxidation is successfully completed.

Another capability that BPFL lacks is a mechanism to specify parallel operations, as seen in ALPS [36]. Such a capability does not appear to be essential in an SPR, although occasionally parallel-processing is done on an ad-hoc basis, especially in photolithography. Implementation of parallelism is nontrivial, but it would be a good extension to BPFL.

A complete implementation of Wood's SECS server [44] for at least one piece of fabrication equipment is necessary to discover and correct shortcomings in the equipment specification part of the language.

Finally, BPFL could benefit from the addition of in-process and in-situ control loops to the language. We envisage high-level constructs like those developed for rework and constraints. The design of such constructs is best carried in concert with research groups familiar with the salient features of control loops.

BPFL is a relatively untested language whose features have undergone several major revisions. While we are confident that the current version of BPFL is appropriate for the tasks it is designed to perform, only further experience with the language will tell what other extensions and revisions are necessary.

The most significant problems with the current system are the user-interface to BPFL and the computer resources required to run the WIP system. The only representation of BPFL is textual. While programmers have few problems with the syntax of BPFL, process engineers are not programmers, and they have no desire to learn programming. Consequently, a graphical user-interface for the specification of BPFL programs is required.

The speed of the WIP system implemented in Common Lisp is adequate for low-volume use. In a high-volume fab it is likely that greater speed would be required. An obvious way to improve the speed of the system is to re-implement the interpreter in C or C++ for commercialization. Common Lisp is an excellent prototyping environment, and had C++ been used for the prototype implementation the language could not have been developed as rapidly as it has been. However, working from the Common Lisp implementation, it should be relatively easy to reimplement the interpreter in C++, and the time required to evaluate a BPFL process flow could be reduced by several orders of magnitude. Furthermore, the code size of the WIP interpreter should decrease by

about 50%. The time required to save and restore run state in the database should not be significant if a high-performance database server is used.

The UI process has several shortcomings. The biggest problems are that it needs a graphical user-interface and a more sophisticated log browsing interface, like the CIM browser application-specific query interface [61]. This would greatly improve the accessibility of the information stored in the WIP-log. Lastly, an object-oriented database would simplify the development of an interface to specify queries on structured data types such as units. For example, this would make it possible to select all `sonogage` records with measurements between a certain range, which is impossible with the current implementation.

Implementation of a scheduling interpreter based on BLOCS [6] is important in order to ensure that BPFL can express the information required for scheduling. A BPFL interface to a CAD database such as OCT [46] would greatly improve the usefulness of masks and layers. For example, it would make possible the automatic calculation of area open for etching, an important factor for determining etch rate in load-sensitive etch processes.

Equally important if BPFL is to be accepted by a user community is that the usefulness of BPFL be proven in a production environment. A test that would prove invaluable in determining the worth of the language and its WIP system would be transferring a process from one fab to another fab, rewriting only the facility-specification code.

References

[1] W. C. Holton et. al., *Japanese Technology Evaluation Program Panel Report on CIM and CAD for the Semiconductor Industry in Japan*, Science Applications International Corporation, (McLean VA) , Dec. 1988.

[2] E. Sachs, S. Ha, A. Hu, A. Ingolfsson and R. Guo, "Run by Run Process Control," Talk given at 1990 SRC/DARPA IC-CIM Workshop, (Berkeley, CA), Aug. 1990.

[3] 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.

[4] 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.

[5] A. S. Wong, "An Integrated Graphical Enviroment for Operating IC Process Simulators," Electronics Research Lab. Memo 89.67, U.C. Berkeley, May 1989.

[6] R. Glassey, "An Overview of BLOCS/M: The Berkeley Library of Objects for Control and Simulation of Manufacturing," *1989 DARPA/SRC Workshop on Integrated Factory Management for Integrated Circuits (IFM-IC)*, (College Station, TX), pp. 81–94, Nov. 1989.

[7] 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.

[8] *Distributed Ingres Manual*, Ingres Corp, Alameda, California, June 1989.

[9] M. R. Stonebraker, E. Hanson and C. H. Hong, "The Design of the POSTGRES Rules System," *IEEE Conf. Data Engineering*, Los Angeles, CA, Feb. 1987.

[10] M. R. Stonebraker and L. A. Rowe, "The Design of POSTGRES," *Proc. 1986 ACM-SIGMOD Conf. on Managment of Data*, (Washington, DC), June 1986.

[11] P. A. Bernstein, "Transaction Process Monitors," *Comm. of the ACM*, vol. 33, no. 11, pp 75–86, Nov. 1990

[12] D. C. Mudie and N. H. Chang, "FAULTS: An Equipment maintenance and Repair System," *Proc. 1990 IEEE/CHMT International Electronics Manufacturing Technology Symposium*, (Washington, DC), Oct. 1990.

[13] C. J. Date, *An Introduction to Database Systems (Volume II)*, (Reading, MA), Addison-Wesley, 1984.

[14] W. G. Oldham, A. R. Neureuther, Y. Shacham and F. Dupois, "Berkeley CMOS Process: A User Guide," Electronics Research Lab. Memo 84.26, U.C. Berkeley, Oct. 1984.

[15] *CAM Systems for Smart Shop Control*, Consilium, Mountain View, California, 1986.

[16] *The PROMIS System: Controlling the Journey to Factory Automation.* Promis Systems Corp., Toronto, Canada, 1987.

[17] A. Aho, R. Sethi and J. Ullman, *Compilers: principles, techniques and tools*, Addison-Wesley, Reading MA, 1986.

[18] H. L. Ossher and B. K. Reid, "Fable: A Programming-Language Solution to IC Process Automation Problems," *ACM-SIGPLAN Notices 18*, no. 6, pp 137–148, Jun. 1983.

[19] H. L. Ossher and B. K. Reid, "Specification for Manufacturing," *Proc. 2nd Annual IC Assembly Automation Conference*, Jan. 1986.

[20] D. S. Boning and M. B. McIlrath, *Guide to the Process Flow Representation Version 2.0*, unpublished report, Aug. 1990.

[21] 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.

[22] J. P. Dishaw and J. Y. Pan, "AESOP – A simulation-based knowledge system for CMOS process diagnosis," *IEEE Trans. Semiconductor Manufacturing*, vol. 2, no. 3, pp 94–103, Aug. 1989.

[23] J. S. Wenstrand, H. Iwai, and R. W. Dutton, "A Manufacturing-oriented Environment for Synthesis of Fabrication Processes," *Proc. 1989 ICCAD Digest of Technical Papers*, pp 376–379, Nov. 1989.

[24] J. S. Wenstrand, W. T. Wong and R. W. Dutton, "Simulation-based Process Specification," *Proc. SRC Techcon '90* Conf., (San Jose, CA), pp 451–454, Oct. 1990.

[25] Voorhees, E. M., "A Work-In-Progress Tracking System for Experimental Manufacturing," *Proc. Second Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering*, (Gaithersburg, MD), pp 190–197, Oct. 1989.

[26] D. Wolfson, *Personal Communication.*, Siemens Corporate Research, Princeton, NJ, Nov. 1990.

[27] K. Funakoshi and K. Mizuno, "A Rule-Based VLSI Process Flow Validation System with Macroscopic Process Simulation," *IEEE Trans. on Semiconductor Manufacturing*, vol. 3, no. 4, pp 239–246, Nov. 1990.

[28] Y. Descotte and J. C. Latombe, "GARI: A Problem Solver that Plans how to Machine Mechanical Parts," IJCAI No. 7, pp 766–772, Aug. 1981

[29] A. Costa and M. Garetti, "Design of a Control System for a Flexible Manufacturing Cell," *Journal of Manufacturing Systems*, vol. 4, no. 1, Jan. 1984

[30] K. K. Lin and C. J. Spanos, "Statistical Equipment Modeling for VLSI Manufacturing: an Application for LPCVD," *IEEE Trans. on Semiconductor Manufacturing*, vol. 3, no. 4, pp 216–229, Nov. 1990.

[31] S. Adiga, *Personal Communication*, IEOR Dept., University of California Berkeley, Dec. 1990.

[32] C. H. Chang and M. A. Melkanoff, *NC machine programming and software design*, Prentice Hall, New York, 1989.

[33] O. Z. Maimon, "A Generic Multirobot Control Experimental System," *Journal of Robotic Systems*, vol. 3, no. 4, pp 451–466, Sep. 1986.

[34] A. W. Naylor and R. A. Volz, "Design of Integrated Manufacturing System Control Software," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 17, no. 6, pp 881–897, Nov. 1987.

[35] S. A. Ray, "A Modular Process Planning System Architecture," presented at *IEE Integrated Systems Conf.*, (Atlanta, Ga), Nov. 1989.

[36] B. A Catron and S. R. Ray, "ALPS – A Language for Process Specification," submitted to *International Journal of Computer Integrated Manufacturing*.

[37] C. B. Williams, *A Process-Flow Specification Language for Manufacturing Semiconductor Integrated Circuits*. Ph.D. thesis, University of California at Berkeley, in preparation.

[38] M. R. Pinto, C. S. Rafferty, and R. W. Dutton, "PISCES II: Poisson and continuity equation solver," Stanford University, Integrated Circuits Lab, Tech. Rep., Sept. 1984.

[39] G. L. Steele, *Common Lisp: The Language*, second edition, Digital Press, 1990.

[40] 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.

[41] L. A. Rowe, *Process-Flow Workshop Report*, unpublished report, Oct. 1990.

[42] R. Hartzel, *Personal Communication*, Texas Instruments, Dallas, TX, Jan.1989.

[43] S. Tang and E. Wood, "An Object-Oriented Design Toolkit for CIM," presented at *1990 SRC/ DARPA IC-CIM Workshop*, (Berkeley, CA), Aug. 1990.

[44] E. J. Wood, H. Schenck and J. Wijaya, "Networking and Object-Oriented Coding for SECS Communication," *Proc. Automated IC Manufacturing Symp.*, Fall Electrochemical Society Meeting, Oct. 1987.

[45] S. G. Duvall, "An Interchange Format for Process and Device Simulation," *IEEE Trans. on CAD*, vol. 7, no. 7, pp 741–754, Jul. 1988.

[46] R. L. Spickelmeir, P. Moore, and A. R. Newton, *A Programmer's Guide to Oct.*, Electronics Research Lab. Memo, U.C. Berkeley.

[47] J. L. Mohammed, *Common Lisp Implementation of SECS II Protocol*, Schlumberger Technologies, July 1990.

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

[49] B. Becker, D. Mudie and L. A. Rowe, "A Paper-Free Replacement for an Engineer's Laboratory Notebook," Talk given at 1990 SRC/DARPA IC-CIM Workshop, (Berkeley, CA), Aug. 1990.

[50] *Ingres ABF/4GL Reference Manual*, Ingres Corp, Alameda, California, June 1989.

[51] W. F. Tichy, "RCS – A System for Version Control," *Software – Practice and Experience*, vol. 15, no. 7, pp. 637–654, July 1985.

[52] T. J. Teorey, *Database Modeling and Design: The Entity-Relationship Approach*, Morgan Kaufmann, San Mateo CA, 1990.

[53] *Using Ingres Through Forms and Menus*, Ingres Corp, Alameda, California, June 1989.

[54] D. Charness and L. A. Rowe, "CLING/SQL – Common Lisp to Ingres/SQL Interface," Electronics Research Lab. Memo 90.40, U.C. Berkeley, May 1990.

[55] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, (Englewood Cliffs, NJ), Prentice-Hall, 1978.

[56] *Allegro COMMON LISP User Guide*, Franz Inc., Berkeley, California, Dec. 1989

[57] B. Stroustrup, *The C++ Programming Language*, (Reading, Massachusetts), Addison-Wesley, 1986.

[58] M. E. Lesk, *Lex - A Lexical Analyzer Generator*, Computing Science Technical Report No. 39, 1975, AT&T Bell Laboratories, Murray Hill NJ 07974.

[59] S. C. Johnson, *Yacc: Yet Another Compiler Compiler*, Computing Science Technical Report No. 32, 1975, AT &T Bell Laboratories, Murray Hill NJ 07974.

[60] S. I Feldman, "MAKE – A Program for Maintaining Computer Programs," *Software – Practice and Experience*, April 1989.

[61] B. C. Smith and L. A. Rowe, "An Application-Specific Ad Hoc Query Interface," Electronics Research Lab. Memo 90.106, U.C. Berkeley, May 1990

[62] Franz Inc, *Common Lisp: The Reference*, Digital Press, 1989.

Appendix A
BPFL Language Reference Manual[1]

A.1 Introduction

This document is a reference manual for the Berkeley Process-Flow Language (BPFL). It describes the current version of BPFL for people who want to code processes or write interpreters.

Specifications written in BPFL describe the fabrication of semiconductor devices. Fabrication requires many processing steps to be performed in a fixed sequence. The word *process* is commonly used to describe two different aspects of fabrication. The first usage refers to the entire sequence of operations that result in a working device (e.g., a "0.35 μm CMOS process"). The second usage refers to an operation within the sequence (i.e., a processing step), usually associated with one piece of equipment (e.g., an oxidation process, an implantation process, or a photolithography process).

A BPFL *process* is composed of a sequence of smaller processes. A complete process specification, like "0.35 μm CMOS," is called a *process flow*. In a BPFL process-flow specification, processes are represented by *procedures*.

The fabrication unit from a process design viewpoint is a *wafer*. In practice, processing is performed on a set of wafers, called a *lot*. During a processing step, the wafers in a lot may be treated serially or in parallel, depending on the equipment used. A lot and its controlling process specification is called a *run*. In a typical fabrication facility, several runs, each at a different stage of completion, will be under the control of one process specification. Within a run a lot may be divided into sub lots, each of which can be processed separately from the rest.

The intent of BPFL is to represent all information about a fabrication process that will be needed by any application programs during the design, manufacture, and testing of semiconductors. Because different applications need different kinds of information, a process flow actually specifies several different views of a process. Most application programs will be *interpreters* that extract information from a BPFL process flow description according to the needs of the application.

---

[1] This document is based on [1].

BPFL is an extension to Common Lisp. The reference manual for Common Lisp is Steele's *Common Lisp: The Language* [2]. It is expected that BPFL will be used in an environment that relies on a database. Consequently, some of the constructs in the language are used to create database objects that other parts of the language access.

The remainder of this document describes BPFL. It is organized as follows. The next section describes the syntax of BPFL. The third section describes the language semantics. The remaining sections describe BPFL objects and procedures for manipulating objects.

A.2 BPFL Syntax

BPFL has a block-structured syntax similar to Pascal. The block-structured notation is converted to Lisp syntax for execution by a BPFL interpreter.

Lisp has a *functional* representation that is simpler than the typical representation of a block-structured language. All Lisp code is represented as lists A list is enclosed in parentheses and elements in a list are separated by spaces. Lisp code consists of *function calls* and *special forms*. For example:

```
(format t "growth rate = " (log 1.23))
```

is a call to the function `format` that prints messages. This call has three arguments. Each of the arguments is *evaluated* before the function is called. The first two arguments (`t` and `"growth rate = "`) are *self-evaluating*. The result of evaluating a self-evaluating object is the object itself. The third argument, "`(log 1.23)`" is a function call. It is evaluated and the result of the evaluation is passed to `format`. So the function call to format becomes

```
(format t "growth rate = " 0.0899051114)
```

Special forms look just like function calls but they evaluate their arguments differently. For example, the Lisp code:

```
(if (> a b)
  (format t "a is bigger than b"))
```

prints "`a is bigger than b`" if the value of variable `a` is larger than the value of `b`. The arguments to `if` are not evaluated in the standard way, because if they were, the `format` function would be called *before* the `if` statement is executed. The `format` function should not be called unless the expression `(> a b)` is true. So `if` evaluates its first argument and only evaluates the second argument if the first argument it true.

BPFL has functions, called procedures, that correspond to Lisp functions. A BPFL procedure call for the Lisp `format` function call above is:

```
format(t, "growth rate = ", log(1.23))
```

The procedure name is moved outside the parentheses, which now contain a list of the arguments to the procedure. The arguments are separated by commas instead of spaces. As with Lisp, the arguments to BPFL procedures are evaluated before the procedure is called.

Instead of function calls and special forms, block-structured languages use *procedure calls* and *statements*. Procedure calls and statements are called *operations*. Statements use a special syntax involving *tokens*. For example, the BPFL `if` statement corresponding to the Lisp `if` special form above is:

```
if a > b then
  format(t, "a is bigger than b")
end;
```

This operation "reads" more like a natural language rather than a functional representation. The use of tokens (e.g., **`if`** and **`then`**) to delimit parts of the **`if`** special form makes the function of each part obvious. The disadvantage of this approach is that the **`if`** special form has a very different syntax than a procedure call, which makes code parsing more difficult. By convention, tokens are written in boldface. The tokens in the **`if`** statement are: **`if`**, **`then`**, **`else`**, and **`end`**.

BPFL expressions use *infix* notation (e.g., `a > b`) instead of the Lisp functional (or prefix) notation (e.g., `(> a b)`). This is true of all numerical operators (e.g., +, -) and logical operators (e.g., >, <). While infix notation is easier to read, it has the disadvantage that ambiguity can exist in the interpretation of an expression. For example, the expression:

```
b + c * d
```

could mean either `(b + c) * d` or `b + (c * d)`. In Lisp, no such ambiguity exists. The first meaning is written as `(* (+ b c) d)` and the second meaning is written `(+ b (* c d))`. BPFL uses the standard rules of operator precedence to resolve ambiguity in expressions. Parentheses may be used to override the operator precedence rules.

Assignment in Lisp is carried out with the `setf` function. The code fragment

```
(setf c (* a b))
```

assigns the value of `a * b` to the variable `c`. BPFL replaces the assignment special form `setf` with a conventional assignment statement. BPFL code for the same operation is:

```
c := a * b;
```

Note also that BPFL uses semicolons (";") to separate procedure calls and statements. For example, the semicolons in

```
a := d * e;
format(t,"a = ~s~%", a);
```

separate the assignment statement from the procedure call. Separators are not required in Lisp.

Common Lisp uses keywords both as argument names and as self-evaluating symbols. Keywords are symbols that begin with a colon (e.g., `:thickness`, `:direction`). In BPFL, keywords are used only as self-evaluating symbols. Argument names are specified by symbols that end with a colon as described in the section on BPFL semantics.

A.2.1 Notational Conventions

A number of special notational conventions are used in this document. The notation is derived from *Common Lisp: The Reference* [3]. Procedures, variables, named constants and statements are described in entries using a distinctive typographical format. An example of an entry is shown in Figure A-1. The first line of each entry is a header line. On the left-hand side of the header line is the entry name and on the right-hand side is the entry type (i.e., `procedure`, `variable`, `constant`, `definition`, or `statement`). Each entry is followed by a list of one or more of the following sections:

1. *Usage*. This section contains a template showing how the entry is used. Entries for data types do not usually include a Usage section. More information about the syntax of usage sections is given below.
2. *Description*. This section describes the entry. Every entry has a description section.
3. *Examples*. This section contains examples of code illustrating how the entry is used. More information about examples is given below.
4. *See Also*. This section contains references to other entries that either assist in understanding the current entry or have closely-related functionality.

The *usage* section describes the way a statement or procedure is used. Literal code is presented in `courier` (e.g., the name of a procedure). Tokens are printed in **`bold-courier`** (e.g., the **`with-lot`**, **`do`** and **`end`** tokens used in the `with-lot` statement.). *Italic* is used to indicate

placeholders that are replaced with literal BPFL code(e.g., *lot-spec* in `with-lot`). Placeholder names are used for illustration. They are referred to in the *description* section for the entry.

The *usage* section uses the following special characters to represent optional and repeating elements of an entry:

1. Brackets ([]) contain optional arguments; what is inside may appear zero or one times.
2. Braces ({}) parenthesize what they contain, but if followed by a *, the contents may appear zero or more times, and if followed by a +, may appear one or more times.
3. A vertical bar | separates mutually-exclusive choices within braces or brackets.

The *examples* section contains samples of code to illustrate the use of the construct de-

---

`with-lot` [Statement]

Usage

```
with-lot lot-spec do
  {operation;}+
end;
```

Description `With-lot` is used to evaluate operations with the `current` lot set to the lots specified in *lot-spec*. *Lot-spec* is either a quoted symbol that is the name of a lot, a quoted list of lot names, or a variable whose value is a lot.

Examples

```
with-lot 'cmos do
  std-wet-oxidation(time: {9 min}, temp: {1000 degC});
end;
        ⇒ t
with-lot '(cmos nwell) do
  std-nitride-deposition(thickness: {1000 Angstrom});
end;
        ⇒ t
l := lot('product);
        ⇒ #<Lot>
with-lot l do
begin
  measure-oxide-thickness();
  etch-oxide();
end;
        ⇒ t
```

Figure A-1: Example entry.

scribed in the entry. They are intended to show the features and use of the construct and do not necessarily represent the best programming style. The character ⇒ represents *evaluation*. For example, the entry:

```
2 + 5
⇒ 7
```

means that the sample code `2 + 5` evaluates to 7.

Sometimes the result of evaluation is an error. The notation used to indicate an error is the word **ERROR**. For example,

Examples
```
5 / 0
⇒ ERROR
```

Occasionally values are printed, and the notation used to indicate what is printed is the word **PRINTS**. For example,

Examples
```
format(t, "Print some output");
⇒ PRINTS Print some output
```

Sometimes the result of evaluation is a complex structure or *object*. In general, objects are represented by:

#<*object-class* [*attributes*]>

Object-class is the name of the class to which the object belongs. *Attributes* is an optional list of one or more attributes of the object. Attributes are usually shown if they are an aid to identifying the purpose of an object. For example, in the following code fragment:

Examples
```
(lot 'product)
⇒ #<lot :id 5>
```

the result of the evaluation is a `lot` object with `id` of 5.

A.3 Data Types

BPFL supports primitive built-in data types such as integers and constructors that can create composite or structured data types such as lists.

A.3.1 Primitive Data Types

The primitive data types supported by BPFL are all available in Common Lisp. For a further discussion of the data types discussed in this section, see *Common Lisp: The Language*, chapter 2.

`integer` [Data type]

Description The integer data type is intended to represent mathematical integers.

Examples `0, 1, +256, -10000`

`ratio` [Data type]

Description The ratio data type allows the exact representation of numbers such as *one-third.* The denominator must be strictly positive and have no leading sign.

Examples `1/2, 1/3, -7/8, 15/17`

`float` [Data type]

Description BPFL supports the single and double-precision formats described in *Common Lisp: The Language.*

Examples `0.0, 0E+0, 3.13d-5, 3.01e-1, -0.0001e+9.`

`boolean` [Data Type]

Description The boolean data type has two possible values `t` and `nil`, corresponding to the Boolean values *true* and *false* respectively.

`symbol` [Data type]

Description A symbol is a sequence of printable characters without any special delimiters. The characters may be alphanumeric or any character in the string "`-*/@$%^?!&_<>.~`". Symbol names may have any number of characters. Anything that cannot be interpreted as a number is a symbol. Symbols are used to name variables and procedures.

The `quote` function can be used to return the symbol itself, rather than the result of evaluating the symbol. `Quote` can be abbreviated to the straight quote character (`'`)

Examples `oxidation, x, wombat, std-wet-oxidation, lumberjack`
See Also `quote`

`keyword` [Data type]

Description A keyword has the same syntax as a symbol, except that the first character of its print representation is a colon (" : "). Keywords differ from symbols in that they self-evaluate, meaning that the value returned by a keyword is the keyword itself.

Examples `:oxidation, :zorkmid, :thickness, :qbd`

`string` [Data type]

Description A string is a sequence of printable characters delimited by double quotes (" "). A printable character is any non-control character in the ASCII character set. If the back-

slash character ("\") appears in a string, the following character is included in the string, even if that character is a backslash or a double quote. The case of characters is significant. Control characters (including newlines) appearing in strings are ignored.

Examples `"string", "\"Hi!\"" "I'm a lumberjack and I'm ok"`

A.3.2 Data Type Constructors

These types have structures and print representations that include other data types.

`list` [Data type]

Description A list is printed as a left parenthesis ("`(`") followed by zero or more values of valid BPFL types, followed by a right parenthesis ("`)`"). Elements of the list are separated from one another by commas ("`,`"). Newlines between elements are ignored. The empty list ("`()`") is synonymous with the symbol `nil`. Parentheses need not be delimited by spaces.

Examples `(a, b, c), (1,2,3), (), (a,(b,c)) ((a))`

`complex numbers` [Data type]

Description BPFL represents complex numbers in cartesian form, with non-complex real and imaginary parts. Parts may be `integer`, `rational` or `float`.

Complex numbers are represented by the characters `#C` followed by a list of the real and complex parts.

Examples `#C(1, 2), #C(1.2, 5.6), #C(1/2, 1/2)`

`unit` [Data type]

Description A unit value consists of a magnitude (an integer, ratio, floating point, or complex number) and a unit expression (a symbol). The units supported by BPFL are based on Système International (SI). Si uses seven *base* units: `m` (meter), `kg` (kilogram), `s` (second), `A` (ampere), `K` (kelvins), `cd` (candela), and `mol` (mole). BPFL contains definitions for the seven base units and a large number of *compound* units derived from these base units. Compound units are specified by a magnitude and a unit expression For example, `1 angstrom` is 1.0×10^{-10} m. Units multiplied together can be separated by a dash ("`-`") or an asterisk ("`*`"). For example, "ohm-m" (a unit of resistivity) is `ohm-m`. The unit designator should contain one slash ("`/`") to separate the numerator and denominator units. For example,

"meters per second" is `m/s`. If there is no numerator part, (e.g., "per second"), the unit designator begins with a slash (e.g., `/sec`). Unit *prefixes* are letters that can be typed in front of a unit name to indicate powers of ten. For example, `mA` means "milliampere" (10^{-3} A). All SI unit prefixes are supported except μ which is represented by `u`. Unit *exponentiation* is represented by a unit followed by a circumflex ("^") and a real number exponent. The unit "square meter" is represented as `m^2`. Parentheses can be used to group units for exponentiation and division (e.g., `kg*m^2/(A*s^3)`). A unit value is printed as {*number unit-expression*}. Two unit-expressions that have the same dimensionality (i.e., that have identical representations in base units) are *dimensionally consistent*. For example, `A*ohm` and `V` have the same base unit representation (i.e., `kg*m^2/(A*s^3)`) and are dimensionally consistent. Dimensionally consistent units can be compared, added and subtracted.

Examples
```
{1000 A}, {20 um}, {10.1 Mohm}, {2.99792e10 cm/s},
{0.0259 V}, {1.05458e-34 J-s}
```

`interval` [Data type]

Description An interval value consists of two numbers or two dimensionally-consistent unit values. Complex number interval types are not supported. The first item in an interval should be less than or equal to the second item. An interval is printed as [*item-1*, *item-2*]. The comma separating the items is optional, but is always printed on output.

Examples
```
[0, 1/2], [-1.0, 1.0], [{10 um}, {20 um}]
```

`object` [Data type]

Description An object contains one or more named subparts called slots. Each slot may contain a value with a different type. Objects may be assigned to variables, passed as arguments to procedures, or returned as the result of a procedure. An object belongs to a *class*, which defines the *slot* names and *method* names. If a class defines a particular slot, then each *instance* of that class (i.e., each object belonging to that class) can store a value under the name of that slot. Certain procedures, called *access procedures*, return values stored in the slots of an object.

Methods are procedures. Several related classes will use the same method name, but each will bind a different procedure to that name. Thus, when the method name is invoked on an object, the procedure used will be determined by the class of the object. Many, but

not all, BPFL objects are stored in a database. Some predefined classes and objects are supplied in the language.

Objects have the following print representation:

#<*object-class* [*attributes*]>

Object-class is the name of the class to which the object belongs. *Attributes* is an optional list of one or more attributes of the object. Attributes are usually written if they are an aid to identifying the purpose of an object. The print representation of an object is not generally *readable*, meaning that the print representation does not normally contain enough information to allow BPFL to reconstruct the object from the print representation. All objects have a `print-object` method defined on them.

Examples Make a PIF snapshot

```
ss := bare-silicon-wafer(resistivity: [{100 ohm-cm} {1000 ohm-cm}],
                         dope: 'n, orientation: '100,
                         quality: 'product);
      ⇒ #<Snapshot>
```

Make a wafer object

```
w := make-wafer(scribe: "MONTY", snapshot: ss);
      ⇒ #<Wafer :id 1>
format(t, "Wafer is ~a", w);
      ⇒ Wafer is #<Wafer :id 1>
format(t, "Wafer is ~a", wafer-scribe(w));
      ⇒ PRINTS Wafer is MONTY
```

See Also `snapshots, segments, attributes, classes and methods`

A.3.3 Classes and Methods

BPFL supports user-defined classes and methods.

`defclass` [Definition]

Usage

```
defclass name
  [docstr]
  [inherits (superclass-list)]
  slots {[classvar] name [:= expr];}+
end;
```

Description `Defclass` defines a class called *name*. *Docstr* is an optional documentation string. Any number of superclasses from which this class inherits may be specified. Inheritance precedence is determined by the order in which the *superclass-list* names are supplied. Any number of slots may be defined. Each slot has a *name* and an optional initial value *expr* that is used to set the value of the slot if it is not specified at the time of class

instance creation. If the **classvar** keyword is specified for a slot, that slot is a class variable for the class (i.e., the slot has the same value for all instances of the class). Slots are accessed by an accessor function whose name is the concatenation of the class name with the slot name, separated by a hyphen. For example, in a class named moscv a slot named cmin can be accessed by the moscv-cmin function.

Defclass defines two default methods for every class. Print-object prints a nonreadable version of a class instance, and make-instance creates a class instance.

Examples

```
defclass moscv
  "MOS capacitance voltage data"
  slots na; /* silicon doping conc */
        dielectric := #m(oxide); /* insulator material */
        measured; /* measured (i,v) pairs */
        frequency; /* Frequency of measurement */
end;
        ⇒ t
x := make-instance('moscv, na: {1.0e13 /cm^2});
        ⇒ #<moscv>
moscv-dielectric(x);
        ⇒ #<material oxide>
```

---

defmethod [Declaration]

Usage

```
defmethod name (class-name, arg-list)
[let {var [:= expr];}+]
begin
  {operation;}+
end;
```

Description Defmethod defines a method called *name* on the class named *class-name*. The method takes the arguments specified in *arg-list*. During the method execution, the class instance on which the method is executed is bound to the local variable self.

Examples

```
defmethod measure (moscv, frequency: = {1 Hz});
begin
  with-equipment e of-type 'cvprobe-station do
    moscv-measured(self) :=
      run-recipe(e, 'measure-cv, frequency: frequency);
  end;
moscv-frequency(self) := frequency;
end;
x := make-instance('moscv, na: {1.0e13 /cm^2});
        ⇒ #<moscv>
measure(x, frequency: {1 kHz});
        ⇒ {1 kHz}
moscv-measured(x)
```

```
⇒ #<array>
```

A.4 Program Structure

Statements are BPFL constructs that control the order of code execution. Every BPFL statement corresponds to a Common Lisp function call, special form or macro that is created when the BPFL code is parsed to Lisp. The rest of this section introduces each of the BPFL statements.

---

constrain [Statement]

Usage

```
constrain
  {operation;}+
  {when expr do
    {operation;}+
  end;}+
end;
```

Description Executes operations in a context where various constraints are enforced. If the constraint is violated, the operations in the appropriate **when** clause are executed.

Examples

```
/* A fatuous example */
constrain
  wet-oxidation(time: {40 min}, temperature: {1000 degC});
  sleep({1 day}); /* Stop the run for one day */
  pattern(mask-name: 'NWELL);

  when max-time-between('wet-oxidation, 'pattern, {1 hr}) do
    halt-run("Constraint violated!");
  end;
end;
```

```
⇒ PRINTS Constraint violated!
```

---

for-each [Statement]

Usage

```
for-each var in list do
  {operation;}+
end;
```

Description Executes the operations once for each element of *list*. During the first iteration of the operations, *var* is bound to the first element of *list*. During subsequent iterations, *var* is bound to subsequent elements.

Examples

```
let result := nil;
begin
  for-each x in '("abc", "fumble", "test") do
    push(length(x), result);
  end;
  result;
end;
```

```
⇒ (4, 6, 3)
```

handler-case [Statement]

Usage

```
handler-case
  {operation;}+
  {on-exception var := condition-type do
    {operation;}+
  end;}
end;
```

Description `Handler-case` executes statements in a context where various specific exception-handlers are defined. If during the execution of the operations a condition is signalled for which there is an appropriate exception clause defined (i.e., the condition has type *condition-type*), then the exception-handler for that condition is executed. During execution of the handler, the variable *var* is bound to the condition.

Examples

```
defcondition tylan-error((), program, step-number);
        ⇒ #<condition>
handler-case
  raise-exception('test, program: "SWETOXB", step-number: 10);
  on-exception c := test do
    format(nil,"Error during furnace run, recipe ~a, step ~a."
           tylan-error-program(c), tylan-error-program(c));
  end;
end;
        ⇒ "Error during furnace run, recipe SWETOXB, step 10."
```

See Also `defcondition`

if [Statement]

Usage

```
if expression then
  {operation;}+
[else
  {operation;}+]
end;
```

Description Evaluates and returns the result of the operations following **then** if *expression* is true, otherwise evaluates and returns the result of the operations following **else**.

Examples

```
nominal := [1, 2];
if interval-p(nominal) then
  make-interval(0.5 * interval-min(nominal),
                2.0 * interval-max(nominal))
else
  make-interval(0.5 * nominal, 2.0 * nominal);
end;
        ⇒ [0.5, 4]
```

let [Statement]

Usage

```
let {var := expr;}+
begin
  {operation;}+
end;
```

Description Let is used to evaluate operations within the context of specific variable bindings. Let returns the value returned by the last operation. Any number of variable bindings may be established. *Var* is the name of the variable to be bound, and *expr* is an expression that is evaluated to establish the value of *var*.

Examples

```
let t := {0.1 um};
    etch-rate := {10 nm/min};
    etch-time := nil;
begin
  etch-time := oxide-thickness / oxide-etch-rate;
  format(nil,"Etch time is ~s",etch-time);
end;
        ⇒ PRINTS "Etch time is {10 min}"
```

rework-loop [Statement]

Usage

```
rework
  {operation;}+
  [rework-test expr]
  [rework-prefix operation]
  [retry-count expr]
  [retry-failure operation]
end;
```

Description Executes operations in a context where the operations, called the rework *body*, may be executed multiple times. Rework has four optional clauses:

1. **Rework-test** is a procedure that is called after each execution of the rework body.
2. **Rework-prefix** is an operation that is executed before the rework body is executed.
3. **Retry-count** is an integer expression that returns the maximum number of iterations of the rework body.
4. **Retry-failure** is an operation that is executed if the **retry-count** is exceeded.

Before the rework body is executed, the **retry-count** expression is evaluated. Then the

rework body is executed. Next, the **rework-test** expression is evaluated. If it returns `nil`, the rework loop is terminated. If it returns any other value, the following actions occur: First, **retry-count** is decremented. If it is less than zero, the **retry-failure** operation is executed; otherwise, the **rework-prefix** operation is executed, and then the rework-body operation is executed again. The whole process repeats from the **rework-test** evaluation.

A program can force a rework by raising a `rework` exception.

Examples

```
let i := 1;
rework-loop
  format(t, "i = ~s,");
  i := i + 1;
  rework-test i < 3
  retry-count 10
end;
        ⇒ PRINTS i = 1, i = 2
```

`step` [Statement]

Usage

```
step name do
  {operation;}+
end;
```

Description `Step` names the operations enclosed within it for documentation purposes.

Examples

```
/* step-path() returns a string that contains the current
   step path for the flow */
step 'NWELL do
  step 'IMPLANT do
    step-path();
 end;
end;
        ⇒ "NWELL/IMPLANT"
```

See Also `step-path`

`view` [Statement]

Usage

```
view viewspec do
  {operation;}+
end;
```

Description `View` is a shorthand version of `viewcase` useful when only one view needs to be specified.

Examples

```
view simulation do
  /* assign nil lots to test lots */
  lot('NWELL) := nil;
```

```
    lot('NCH) := nil;
  end;
```

viewcase [Statement]

Usage

```
viewcase
  {when viewspec do
    {operation;}+
  end;}+
end;
```

Description `Viewcase` is used to control the execution of sections of BPFL code depending on what capabilities an interpreter possesses. A *viewspec* is a predicate of views.

Examples

```
viewcase
  when fabrication do
    develop-resist(mask-name: mask-name);
  end;
  when simpl do
    simpl-op("DEVL","ERST");
  end;
end;
```

See Also `view`

while [Statement]

Usage

```
while expr do
  {operation;}+
end;
```

Description Executes the operations until *expr* returns `nil`. If *expr* returns `nil` when first evaluated, the operations will not be executed.

Examples

```
let i := 4;
    j := 0;
begin
  while i > 1 do
    j = j + i;
    i = i - 1;
  end;
  format(t, "j = ~s", j);
end;
        ⇒ PRINTS j = 9
```

with-equipment [Statement]

Usage

```
with-equipment var of-type equipment-spec do
  {operation;}+
end;
```

Description `With-equipment` is used to execute operations with an allocated piece of

equipment. An object representing a piece of equipment satisfying *equipment-spec* is assigned to *var*. The equipment is allocated before the operations are executed and deallocated after they are completed. *Equipment-spec* is a symbol representing the name of a piece of equipment to be allocated.

Examples Assume that a spinner called `y1-mti-spinner` has been defined.

```
with-equipment e of-type 'spinner do
  format(nil, "allocated ~s", e);
end;
        ⇒ PRINTS "allocated #<equipment :name y1-mti-spinner>"
```

`with-lot` [Statement]

Usage

```
with-lot lot-spec do
  {operation;}+
end;
```

Description `With-lot` is used to evaluate operations with the `current` lot set to the lots specified in *lot-spec*. *Lot-spec* is either a quoted symbol that is the name of a lot, a quoted list of lot names, or a variable whose value is a lot.

Examples

```
with-lot 'cmos do
  std-wet-oxidation(time: {9 min}, temp: {1000 degC});
end;
        ⇒ t
with-lot '(cmos nwell) do
  std-nitride-deposition(thickness: {1000 Angstrom});
end;
        ⇒ t
l := lot('product);
        ⇒ #<Lot>
with-lot l do
begin
  measure-oxide-thickness();
  etch-oxide();
end;
        ⇒ t
```

A.4.1 Definitions and Declarations

Definitions and declarations supply information about a program. They are used to specify entities in a facility, global variables and constants, and process procedure libraries in a process flow.

`defglobal` [Definition]

Usage `defglobal` *name* `:=` *expr*`;`

Description `Defglobal` is used to define a global variable. *Name* is a symbol that names the variable. *Expr* is evaluated and the value returned by it is stored in the variable.

By convention, the names of global variables begin and end with asterisks ("*"). The use of global variables in discouraged.

Examples

```
defglobal *resist-thickness* := {1.0 um};
        ⇒ {1.0 um}
*resist-thickness* := {5 km};
        ⇒ {5 km}
```

See Also `defconstant`

`defconstant` [Definition]

Usage `defconstant` *name* `:=` *expr*`;`

Description `Defconstant` defines a global constant. *Name* is a symbol that names the constant. *Expr* is evaluated and the value returned by it is the constant.

By convention, the names of global constants begin and end with asterisks ("*"). It is illegal for a user to attempt to change the value of a constant defined with `defconstant`, or to attempt to redefine it using another `defconstant`.

Examples

```
defconstant *permittivity* := {8.85418e-14 F/cm}
        ⇒ {8.85418e-14 F/cm}
*permittivity* := 12.5;
        ⇒ ERROR
defconstant *permittivity* (1 / (*permeability* * (*c*)^2));
        ⇒ ERROR
```

See Also `defglobal`

`require` [Declaration]

Usage `require(`*library-name*`, [version:` *version*`]);`

Description `Require` is used to indicate that a BPFL code module uses the procedures and declarations in a code library. *Library-name* is the name of the library to use. *Version* is a string or symbol that indicates what version of the library to use. If *version* is a string containing an RCS revision number or a revision tag, that version of the module is used. If *version* is a symbol, the latest version of the module is used and the symbol determines what action BPFL will take if a new version of the module is created. The allowable symbols are:

1. `Static` - the library code is never updated,

2. `Latest` - the library code is always updated whenever a new version is checked in.
3. `Query` - whenever a new version of the library is checked in, the user is asked if the library should be updated.

Examples Use version 1.1 of the `litho library`.

```
require(litho, version: "1.1");
```

Use the `ucb-std` library with version tag "contact."

```
require(ucb-std, version: "contact");
```

Use the latest version of `litho` and query the user for updates

```
require(litho, version: query);
```

A.4.2 Procedure Calls

Procedure calls are represented as a symbol followed by a list of arguments enclosed in parentheses. The symbol is the name of the procedure to be called. The list comprises the actual arguments to the procedure. There are two kinds of arguments: *positional* and *named*. For positional arguments, each element in the actual argument list corresponds to an argument in the formal argument list. The order of the elements of the list is significant when assigning actual values to the formal arguments of the procedure being called. For named arguments, the elements of the list are used in pairs. The first member of a pair must be a name. The second member is the argument value. The argument value is passed as the named argument. The order in which arguments are given is not significant because formal arguments are assigned by name. It is an error for a named argument to have a name and no value.

If both positional and named arguments are to be passed to a procedure, then the positional arguments must appear first in the list.

Examples

```
lot-name(l); /* one positional argument */
set-union('(a,b,c), '(d,e,f)); /* two positional arguments */
  /*  two named arguments */
std-wet-oxidation(time: {11 min}, temp: {900 degC});
  /* a positional argument and a named argument */
sort(list, descending: t);
getf(result,:rework); /* two positional arguments */
```

A.4.3 Procedure Definitions

---

defflow [Definition]

Usage

```
defflow name ({arg-list})
  [docstring]
[let {var := expr;}+]
begin
  {operation;}+
end;
```

Description `Defflow` is used to define BPFL procedures. The global symbol *name* is given a procedure definition. *Docstring* is an optional string that is used to document the procedure. Local arguments for the procedure may be defined in the optional `let` statement. Any number of local variable bindings may be established. *Var* is the name of the local variable to be bound, and *expr* is an expression that is evaluated to establish the value of *var*. *Arg-list* is the formal argument list. The *body* of the procedure is given by the operations appearing between the **begin** and **end**.

Examples Here is a procedure that computes powers of two using a recursive algorithm.

```
defflow rpower-of-two (n)
begin
  if n = 0 then
    1;
  else
    rpower-of-two(n - 1);
end;
        ⇒ rpower-of-two
rpower-of-two(3)
        ⇒ 8
rpower-of-two(24)
        ⇒ 16777216
```

A.4.3.1 Argument Declarations

The argument-declaration defines the formal arguments. Positional arguments are specified by symbols. Named arguments are specified by the name of the argument. The `was-supplied` function can be used to tell if a named argument has been passed. `Was-supplied` takes an argument name and returns the value `nil` if the argument was not passed, and `t` if it was.

Named arguments can also have default values. Each named argument variable-specifier has the following syntax:

```
name: { = initexpr }
```

name names the variable. `Initexpr` is an optional *initialization expression* that is used to initial-

Common Lisp	BPFL	Description
`(a b c)`	`(a,b,c)`	Three positional arguments.
`(&key (a 5) b c)`	`(a: = 5,b,c)`	Three named arguments, with a default value of 5 for the first argument.
`(a b c &key test)`	`(a,b,c,test:)`	Three positional arguments and one named argument.
`(&key (test t test-supplied))`	`(test: = t)`	One named argument with a default value and a was-supplied variable.

Table A-1: Comparison between Common Lisp and BPFL argument lists.

ize the variable. Further discussion of the argument-declaration list is found in the section on semantics. Table A-1 contains examples of how Common Lisp and BPFL argument lists differ. BPFL does not support optional positional arguments.

A.4.3.2 Procedure Body

The procedure body is a boundary that defines the scope of local variables and control flow. Local variable names do not affect the user of the same name in other procedure bodies. Control of the execution sequence within the procedure body is restricted to that procedure. Control may not be transferred to another procedure except by calling a procedure, exiting the current procedure body, or raising an exception. The procedure body is composed of statements and procedure calls.

A.5 BPFL Semantics

This section describes the semantics of BPFL that must be preserved by all interpreters. Interpreters can provide additional functionality as long as these basic semantics are not violated.

Each step in a process specifies an operation to be carried out according to the process specification. The interpreter analog to a step is *evaluation*. Evaluation produces a *result value* for each expression and statement. For example, the result of evaluating a variable is the value of the variable. The next three sub-sections describe the behavior of constants, variables and procedure calls when they are evaluated.

A.5.1 Constants

Constants evaluate to the value they denote (i.e., they *self-evaluate*). When a number (e.g., integer, ratio, floating point or complex) is evaluated, the result value is just the number (except that integral valued ratios may be converted to integers during evaluation. Strings, unit values, intervals, and keywords also self-evaluate. The symbols `t` and `nil`, representing the Boolean values true and

false respectively, are the only self-evaluating symbols.

A.5.2 Variables

Any symbol that is not a keyword can have a value. This value is returned when the symbol is evaluated. If the symbol has not had a value assigned to it, an error results. The assignment operator `:=` is used to set the value of a symbol. The `quote` special procedure may be used to return a symbol as the result of evaluation rather than the symbol's assigned value.

There are two classes of variables: *local* and *global*. The formal arguments of a procedure are local variables. Local storage is allocated to hold the value of the actual argument. The same symbol can be used in different procedures to name a local variable. These variables (i.e., storage locations) are distinct. Thus, a symbol has an assigned value for each procedure that it is used in. A value must be assigned in each procedure before the variable can be used.

A.5.3 Procedure Calls

There are two types of procedures that can be called: *built-in* procedures (i.e., those implemented directly by an interpreter, such as `user-dialog`) and *user-defined* procedure (i.e., a procedure defined by `defflow`). The actual arguments to a procedure are determined in the same way for all types of procedures, except for *special* built-in procedures.

The evaluation of a procedure call is composed of the following steps:

1. Evaluate actual arguments,
2. Determinate the procedure to be called,
3. Initialize formal arguments and local variables, and
4. Execute the procedure body.

Each argument is evaluated and the result becomes an argument to the procedure. In the case of named arguments, only the value is evaluated. The division between positional and named arguments is determined before evaluation begins (i.e., it is syntactically determined).

If the procedure name is not recognized by the interpreter as a built-in procedure, it is assumed to be a user-defined procedure. If no user-defined procedure with that name exists, an error is raised.

Once a procedure body has been found for execution, the actual arguments are assigned to the formal arguments of the procedure. BPFL argument passing is almost identical to Common Lisp

argument passing, with one difference. The `was-supplied` function is used to determine if a named variable was passed to a procedure. `Was-supplied` takes the name of an argument and returns `t` if it was passed and `nil` otherwise. Common Lisp *was-supplied* variables perform the same function.

After the formal arguments have been initialized, the local variables defined in the `let` part of the `defflow` are initialized in order of specification. If an initialization expression is given, the operation is evaluated and the result is used to initialize the variable. Otherwise, the variable is initialized to `nil`. Initialization expressions may reference any local variable (or formal argument) that has already been initialized.

Once the arguments and local variables have been initialized, the procedure is evaluated. When evaluation of the procedure body is complete, the procedure returns.

A.5.4 Attributes

Objects in BPFL have attributes. For example, a `mask` has a `location` attribute. Many objects in BPFL form a hierarchy (e.g., materials and equipment). Objects in a hierarchy *inherit* attributes from their *parents* or superclasses. For example, in the material definitions:

```
defmaterial silicon ((), atomic-weight: 28);
defmaterial si ((silicon), monocrystalline: t);
defmaterial si29 ((si), atomic-weight: 29);
```

The `silicon` material has a single attribute `atomic-weight` with a value of `28`. `Si` has two attributes, `atomic-weight` and `monocrystalline`, with values `28` and `t` respectively. `Si29` also has two attributes, but the value for `atomic-weight` is `29`, because the value supplied in the definition for `si29` overrides the value inherited from `si`.

The above example is for *simple* attributes. Simple attributes have just a single value associated with them. For example the integer `28` is the value for the `atomic-weight` attribute for `si`. Attributes can also be *complex*. Complex attributes have attributes attached to them. For example, in the equipment definition:

```
defequipment spinner ((),
  recipes: (spin-on-resist: (frame: spinner),
            strip-resist: (frame: spinner),
            develop-resist: (frame: spinner, spin-dry: t)));
```

`spinner` is an equipment definition with one attribute, `recipes`. `Recipes` has three attributes: `spin-on-resist`, `strip-resist`, and `develop-resist`. Each of these attributes in turn

has attributes. `Spin-on-resist`, `strip-resist` and `develop-resist` all have `frame` attributes, and `develop-resist` also has a `spin-dry` attribute.

Complex attributes inherit recursively. For example, if another piece of equipment is defined:

```
defequipment y1-mti-spinner ((spinner),
  recipes: (spin-on-resist: (program: 1, frame: mti-spin),
            strip-resist: (program: 3),
            develop-resist: (program: 70)));
```

`recipes` still has three attributes (i.e., `spin-on-resist`, `strip-resist`, and `develop-resist`), but each of those attributes has a new attribute named `program`. The `frame` attribute of `spin-on-resist` has the value `mti-spin`.

Suppose a BPFL procedure defines two material instances[1]:

```
m1 := #m(poly nominal-thickness: {100 nm},
         dopant: #m(phosphorus);
m2 := #m(poly nominal-thickness: {200 nm}
         dopant: #m(phosphorus));
```

Comparing the two materials with the = operator returns `nil`, because the materials have different values for the `nominal-thickness` attribute. However, if m2 is defined as

```
m2 = #m(poly dopant: #m(phosphorus))
```

then `m1 = m2` returns `t`, because m1 and m2 have identical values for attributes that both of them possess.

A.6 Wafer State Representation

BPFL maintains wafer-state representation by using a version of the *Profile Interchange Format* (PIF [4]). The state of a wafer is represented by a *snapshot*. Snapshots are composed of three types of objects: *boundaries*, *segments* and *attributes*. Snapshots are also objects.

Attributes have two slots: a *name* and a *value*. The value stored in an attribute can be of any type. Boundaries have two slots: *upper* and *lower*. These contain pointers to segments above and below the boundary, Segments have no slots; they are objects to which attributes and boundaries are attached by reference. The association between segments, attributes and boundaries is done with a *snapshot*. Snapshots have four slots. *Parent* is a pointer to the snapshot that was used in the creation of this snapshot. *Segments* is a list of segments within a snapshot. *Attr-hash* is a hashtable

[1] `#m` is shorthand for the `material` procedure that creates material instances.

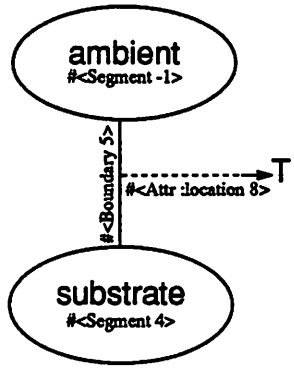

Figure A-2: Bubble diagram for sample wafer.

which contains references to all objects in a snapshot. The key to attr-hash is an object (i.e., a boundary, segment or attribute) and the value returned is a list of attributes and boundaries attached to the object. *Rev-hash* is a hash table that maps attributes is a snapshot to the list of objects that the attributes are attached to.

For example, consider, the snapshot created by the procedure call:

```
ss := bare-silicon-wafer(
        resistivity: [{18 ohm-cm}, {22 ohm-cm}], dope: 'p,
        crystal-face: '100, quality: 'product);
```

Figure A-2 shows the PIF bubble diagram for the wafer state represented by the snapshot stored in the `ss` variable. The `parent` slot of `ss` is `nil`, because this is a new snapshot. It would be non-nil only if the snapshot were a modified version of an older snapshot. This situation arises when some wafers that share the same snapshot are processed differently from other wafers with the same snapshot. Table A-2 contains a description of all of the objects in the snapshot. Every object has an identifier (`id`) that is unique within a given run. For example, the `segment` describing the `ambi-`

Object	Description
`#<Segment -1>`	The `ambient` segment.
`#<Snapshot 2>`	The snapshot describing the wafer.
`#<Attr :origin-step 3>`	An attribute describing the step-path at which the snapshot was created.
`#<Segment 4>`	The `silicon` segment.
`#<Boundary 5>`	The boundary between `silicon` and `ambient`.
`#<Attr :material 6>`	The material attribute attached to the `silicon` segment.
`#<Attr :origin-step 7>`	An attribute describing the step-path at which the `silicon` segment was created.
`#<Attr :location 8>`	The location of the boundary between the `silicon` and `ambient` segments.

Table A-2: Example snapshot objects.

Key	Value
#<Segment -1>	(#<Boundary 5>)
#<Snapshot 2>	(#<Attr :origin-step 3>)
#<Attr :origin-step 3>	nil
#<Segment 4>	(#<Attr :origin-step 7> #<Attr :material 6> #<Boundary 5>)
#<Boundary 5>	(#<Attr :location 8>)
#<Attr :material 6>	nil
#<Attr :origin-step 7>	nil
#<Attr :location 8>	nil

Table A-3: Attr-hash slot contents.

ent has an id of -1. System-defined objects (such as ambient) have negative ids. System-defined objects are maintained by the BPFL interpreter and the user may not alter them. User-defined objects are numbered sequentially from 1 upward. For example, the snapshot object has an id of 2. Every time a snapshot or segment is created, the BPFL interpreter attaches an origin-step attribute to the snapshot or segment. This is done so that the user can see when particular parts of the snapshot were created.

The segments slot contains a list of the segments in the snapshot, in reverse order to which they were created. In this example, segments has the value (#<Segment 4> #<Segment -1>). The contents of the attr-hash slot for the snapshot are shown in Table A-3. For example, the ambient segment has the boundary between the two segments attached to it, because the boundary exists between ambient and another segment. Likewise, the snapshot has an origin-step attribute attached to it.

Table A-4 shows the contents of the rev-hash table. This table contains an entry for every attribute in a snapshot. The entry is a list of the objects to which the attribute is attached. This information can be deduced from the contents of the attr-hash table, but rev-hash makes many PIF queries easier to implement.

Key	Value
#<Attr :origin-step 3>	(#<Snapshot 2>)
#<Attr :material 6>	(#<Segment 4>)
#<Attr :origin-step 7>	(#<Segment 4>)
#<Attr :location 8>	(#<Boundary 5>)

Table A-4: Rev-hash slot contents.

Consider executing the PIF operation:

```
grow-in-lot(#m(substrate), material: #m(oxide),
            nominal-thickness: {100 nm});
```

This operation adds a new segment of oxide above the silicon in the wafer. The new PIF snapshot for the wafer is shown in Figure A-2. Assuming that the grow-in-lot operation is applied to only some of the wafers with the snapshot in Figure A-3, it is necessary to create a new snapshot for those wafers that are to have oxide grown on them. The result of executing grow-in-lot creates a new snapshot #<Snapshot 9>. The parent slot of this snapshot is #<Snapshot 2>. The new snapshot shares the original attributes of the parent snapshot if possible. Table A-5 lists the objects for the new snapshot. Note that the segments and attributes of the parent snapshot are retained wherever possible. For example, the location attribute describing the boundary between substrate and oxide in the new snapshot is the same attribute used in the original snapshot to describe the boundary between substrate and ambient, because oxide will only grow at that boundary. Table A-6 shows the attr-hash slot for the snapshot after the operation.

As an example of querying the snapshot, suppose we wanted to find all oxide segments on the surface of the wafer. Since surface segments have the property of being adjacent to the ambient segment, one approach to solving this query is to get the value of attached boundaries for the ambient segment using the attr-hash table, and to return the list of segments in the lower

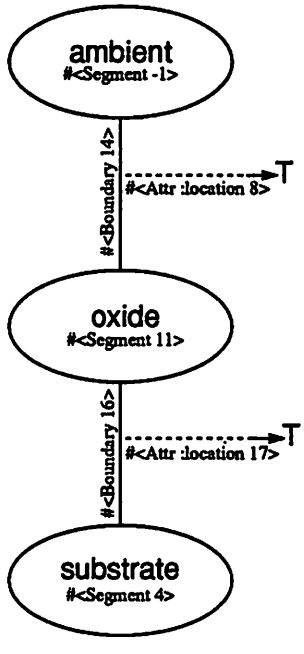

Figure A-3: Bubble diagram for sample wafer after oxide growth.

Object	Description
#<Segment -1>	The ambient segment.
#<Segment 4>	The silicon segment.
#<Attr :material 6>	The material attribute attached to the silicon segment.
#<Attr :origin-step 7>	An attribute describing the step-path at which the silicon segment was created.
#<Attr :location 8>	The location of the boundary between the silicon and oxide segments.
#<Snapshot 9>	The snapshot describing the wafer.
#<Attr :origin-step 10>	An attribute describing the step-path at which the snapshot was created.
#<Segment 11>	The oxide segment.
#<Attr :material 12>	The material attribute attached to the oxide segment.
#<Attr :nominal-thickness 13>	The nominal-thickness attribute attached to the oxi segment.
#<Boundary 14>	The boundary between oxide and ambient.
#<Attr :origin-step 15>	An attribute describing the step-path at which the oxide segment was created.
#<Boundary 16>	The boundary between silicon and oxide.
#<Attr location 17>	The location of the boundary between the silicon and ambient segments.

Table A-5: New snapshot objects.

slot of each boundary. The BPFL procedure `find-surface-segments-in-lot` performs this operation on all snapshots for wafers in the current lot:

```
segs := find-surface-segments-in-lot(ss);
```

`Find-surface-segments` can be instructed to return only segments with particular attributes.

Key	Value
#<Segment -1>	(#<Boundary 14>)
#<Segment 4>	(#<Boundary 16> #<Attr :origin-step 7> #<Attr :material 6>)
#<Attr :material 6>	nil
#<Attr :origin-step 7>	nil
#<Attr :location 8>	nil
#<Snapshot 9>	(#<Attr :origin-step 10>)
#<Attr :origin-step 10>	nil
#<Segment 11>	(#<Boundary 16> #<Attr :origin-step 15> #<Boundary 14> #<Attr :nominal-thickness 13> #<Attr :material 12>)
#<Attr :material 12>	nil
#<Attr :nominal-thickness 13>	nil
#<Boundary 14>	(#<Attr location 17>)
#<Attr :origin-step 15>	nil
#<Boundary 16>	(#<Attr :location 8>)
#<Attr location 17>	nil

Table A-6: `Attr-hash` table for new snapshot.

For example, to find all oxide surface segments:

```
segs := find-surface-segments-in-lot(ss,
                                     material: #m(oxide));
```

The variable segs contains a list of all oxide surface segments. Ordinarily only one such segment would exist, but it is possible to restrict the search to find segments at a particular location using the procedure find-surface-segments-at-location. In general, the user needs to specify enough information about the desired segment to ensure that it is unique if only one segment is required. The procedure pif-attr-val returns the value associated with any pif attribute. So the procedure call:

```
pif-attr-val-in-lot(segs, :nominal-thickness, ss);
```

returns a list of all the nominal-thickness attributes for the surface oxide segments. If only one of the values in the list is required (e.g., the maximum oxide thickness on the surface), a procedure can be used to obtain the desired value (e.g., the max procedure could be used to extract the maximum value from the list).

The remainder of this section lists BPFL procedures for creating, modifying and querying snapshots and PIF objects.

A.6.1 Creation and Manipulation of PIF objects.

---

make-root-snapshot [Procedure]

Usage `make-root-snapshot()`

Description Creates a PIF description of a snapshot with no objects. Objects must be created and attached using other PIF procedures

---

make-segment [Procedure]

Usage `make-segment()`

Description Creates a PIF segment. The segment is not part of any snapshot until explicitly attached using add-segment.

---

make-boundary [Procedure]

Usage `make-boundary(`*seg1*`,` *seg2*`);`

Description Creates a boundary between the two segments *seg1* and *seg2*.

`make-pif-attr` [Procedure]

Usage `make-pif-attr(`*name*`, `*value*`);`

Description Creates a pif attribute with the specified *name* and *value*.

`snapshot-p` [Procedure]
`pif-attr-p` [Procedure]
`segment-p` [Procedure]
`boundary-p` [Procedure]
`pif-object-p` [Procedure]

Usage
```
snapshot-p(arg);
pif-attr-p(arg);
segment-p(arg);
boundary-p(arg);
pif-object-p(arg);
```

Description These procedures return `t` if *arg* is of the appropriate type; otherwise, return `nil`.

For example, `snapshot-p` returns `t` if *arg* is a snapshot. `Pif-object-p` returns `t` if *arg* is a snapshot, pif attribute, segment or boundary.

Examples
```
ss := make-root-ss();
        ⇒ #<snapshot>
snapshot-p(ss);
        ⇒ t
boundary-p(ss);
        ⇒ nil
pif-object-p(ss);
        ⇒ t
```

`add-segment` [Procedure]

Usage `add-segment(`*ss*`, `*seg*`);`

Description Adds the segment *seg* to the snapshot *ss*.

`bind-boundary` [Procedure]

Usage `bind-boundary(`*ss*`, `*b* `[,`*oldb*`]);`

Description Adds the boundary *b* to the snapshot *ss*. If a third argument *oldb*, which must be a boundary in *ss*, is specified, then *b* replaces *oldb* in the snapshot. The segments above and below *oldb* are attached to *b*.

`snapshot-parent` [Reader]
`snapshot-attr-hash` [Reader]
`snapshot-rev-hash` [Reader]
`snapshot-segments` [Reader]

Usage
```
snapshot-parent(ss);
snapshot-attr-hash(ss);
snapshot-rev-hash(ss);
snapshot-objects(ss);
```

Description These procedures return the appropriate slots for the snapshot *ss*.

`snapshot-objects` [Procedure]

Usage
```
snapshot-objects(ss);
```

Description This procedure returns a list of all of the PIF objects in the snapshot *ss*.

`object-in-ss` [Procedure]

Usage
```
object-in-ss(ss, obj);
```

Description Returns `t` if the pif object *obj* is in the snapshot *ss*, otherwise `nil`.

`bound-pif-attrs` [Procedure]

Usage
```
bound-pif-attrs(ss, obj);
```

Description Returns a list of all pif objects in snapshot *ss* that are bound to the pif object *obj*.

`boundary-upper` [Reader]
`boundary-lower` [Reader]

Usage
```
boundary-upper(b);
boundary-lower(b);
```

Description These procedures return the appropriate slots for the boundary *b*.

`pif-attr-name` [Reader]
`pif-attr-value` [Reader]

Usage
```
pif-attr-name(attr);
pif-attr-value(attr);
```

Description These procedures return the appropriate slots for the pif attribute *attr*.

`pif-attr-val` [Acessor]

Usage
```
pif-attr-val(ss, obj, name, [, default]);
pif-attr-val(ss, obj, name) := val;
```

Description This procedure accesses the value of an attribute bound to a pif object.

The first usage returns the value of the attribute named *name* attached to the PIF object *obj* in the snapshot *ss*. If no such attribute exists, the value *default* is returned. If *default* is not supplied and the attribute does not exist, `nil` is returned.

The second usage sets the value of the specified attribute to *val*.

`remove-pif-object` [Procedure]

Usage `remove-pif-object(ss, obj);`

Description This procedure removes the PIF object *obj* from the snapshot *ss*. *Obj* and all attributes attached to it are removed.

A.6.2 Snapshot modification

`etch-segment` [Procedure]
`etch-segment-in-lot` [Procedure]

Usage `etch-segment(ss, s, loc);`

Description `Etch-segment` removes the segment *s* at the location *loc* in the snapshot *ss*.

`lot-snapshots` [Procedure]

Usage `lot-snapshots({lot-name});`

Description Returns a list of snapshots used by wafers in the specified lots. If no lot names are supplied, the `current` lot is assumed.

`segments-in-lot` [Procedure]

Usage `segments-in-lot({lot-name});`

Description Returns a list of all segments in snapshots for wafers in the specified lots. If no lot name is supplied, the `current` lot is assumed.

find-segments	[Procedure]
find-surface-segments	[Procedure]
find-surface-segments-at-location	[Procedure]
find-segments-in-lot	[Procedure]
find-surface-segments-in-lot	[Procedure]
find-surface-segments-at-location-in-lot	[Procedure]

Usage

```
find-segments(ss {,name: value}*);
find-surface-segments(ss {,name: value}*);
find-surface-segments-at-location(ss, loc {,name: value}*);
find-segments-in-lot(name: value {,name: value}*);
find-surface-segments-in-lot(name: value {,name: value}*);
find-surface-segments-at-location-in-lot(loc,
        name: value {,name: value}*);
```

Description These procedures return a list of segments with attributes matching those supplied. Any number of attribute *name*, *value* pairs may be passed to these procedures. `Find-segments` returns all segments in the snapshot *ss* with attributes named *name* and value *value*. `Find-surface-segments` returns all segments in the snapshot with the specified attributes on the surface of the wafer. `Find-surface-segments-at-location` returns all surface segments at the location *loc*.

`Find-segments-in-lot`, `find-surface-segments-in-lot`, and `find-surface-segments-at-location-in-lot` are identical to the above procedures but operate on all snapshots in the current lot.

deposit-in-lot	[Procedure]

Usage

```
deposit-in-lot(loc {,name: value}*);
```

Description This procedure deposits a segment on all wafers in the lot at the location *loc*. The segment will have the specified attributes attached to it.

grow-in-lot	[Procedure]

Usage

```
grow-in-lot(mat {,name: value}*);
```

Description This procedure simulates growing a segment on top of segments with material attribute *mat*.

etch-material-in-lot	[Procedure]

Usage

```
etch-material-in-lot(mat, loc);
```

Description This procedure etches all surface segments of material *mat*. The segments are re-

Slot	Description
snapshot	Pointer to PIF snapshot describing the wafer state.
id	Integer uniquely identifying the wafer in a fab.
index	Integer uniquely identifying the wafer in a run.
scribe	String containing the wafer scribe

Table A-7: Wafer class description.

moved at location *loc*.

A.7 Wafer and Lot Specification

BPFL represents wafers using the wafer class described in Table A-7. Wafer has four slots:

1. snapshot is a pointer to a PIF snapshot describing the wafer (see section A.6.1).,
2. id is an integer that uniquely identifies the wafer in the fab,
3. index is an integer that uniquely identifies the wafer in the run, and
4. scribe is a string containing the wafer name.

Wafers are created using the initialize-wafer procedure. For example,

```
w := initialize-wafer(snapshot: make-root-ss(),
                      scribe: "CMOS-1");
```

creates a wafer with a new snapshot and a scribe of "CMOS-1." Allocate-wafers does not have an index argument, and the index and id slots in the wafer class are generated automatically by the BPFL interpreter. Once a wafer is created, a user may not change any of the values in the slots. Normally, wafers are created using the allocate-lot procedure described below.

Lots are represented using the lot class shown in Table A-8. Lot has two slots:

1. id, which contains a unique integer for the lot, and
2. bits, which is an integer that is used to indicate which wafers are present in the lot.

The least-significant bit in bits is used to represent the wafer with index of 1. The next least-

Slot	Description
id	Integer uniquely identifying the lot.
bits	Integer representing wafers present in the lot.

Table A-8: Lot class description

significant bit is used to represent the wafer with index of 2 and so forth. For example, if the value of bits is 35, the lot contains the wafers with indices 1, 2, and 6 (35 in binary is 100011). Lots are created using the allocate-lot procedure. Every lot is also associated with a *name*, which is a symbol used to name the lot.

---

initialize-wafer [Procedure]

Usage

```
initialize-wafer(scribe: string,
                 snapshot: ss);
```

Description This procedure creates a wafer object with the specified *scribe* and *snapshot* slots.

Examples

```
w := initialize-wafer(scribe: "TEST",
                      snapshot: bare-silicon-wafer());
        ⇒ #<Wafer :id 1>
wafer-scribe(w);
        ⇒ "TEST"
```

---

wafer-id [Reader]
wafer-index [Reader]
wafer-scribe [Reader]
wafer-snapshot [Reader]

Usage

```
wafer-id(w);
wafer-index(w);
wafer-scribe(w);
wafer-snapshot(w);
```

Description These procedures return the values in the id, scribe and snapshot slots for the wafer object *w* respectively.

Examples

```
w := initialize-wafer(scribe: "TEST",
                      snapshot: bare-silicon-wafer());
        ⇒ #<Wafer :id 1>
wafer-snapshot(w);
        ⇒ #<Snapshot :id 20>
wafer-id(w);
        ⇒ 1
```

---

indexed-wafer [Procedure]

Usage

```
indexed-wafer(i);
```

Description This procedure returns the wafer with the index *i*.

Examples

```
indexed-wafer(1);
        ⇒ #<Wafer :id 1>
```

`deallocate-wafer` [Procedure]

Usage `deallocate-wafer(w);`

Description `Deallocate-wafer` removes the wafer *w* from the wafers in a run

`allocate-lot` [Procedure]

Usage
```
allocate-lot(names: lot-name-list,
             sizes: lot-size-list,
             snapshot: ss);
```

Description This procedure allocates wafers and creates lots using the wafers. *Lot-name-list* is a list of symbols that are used to name the lots. *Lot-size-list* is a list of integers giving the size of each of the created lots. *Lot-name-list* and *lot-size-list* must be the same length. The n^{th} element of each list gives the name and size of the n^{th} lot, respectively. *Ss* is a snapshot that specifies the initial state of the wafers. `Allocate-lot` returns a list of the created lots.

`Allocate-lot` is intended to be a high-level interface for users to enter the names of BPFL wafers. Other procedures are provided to move wafers between lots.

Examples
```
allocate-lot(names: '(cmos, nwell, nch),
             sizes: '(20, 1, 1),
             snapshot: bare-silicon-wafer());
⇒ (#<lot :id 1> #<lot :id 2> #<lot :id 3>)
```

`lot-name` [Reader]
`lot-id` [Reader]
`lot-bits` [Reader]

Usage
```
lot-name(lot);
lot-id(lot);
lot-bits(lot);
```

Description These procedures return the respective attribute for the lot object.

`create-lot` [Procedure]

Usage `create-lot(symbol);`

Description `Create-lot` allocates a new empty lot named *symbol*. It is an error if a lot with the name *symbol* already exists.

Examples
```
create-lot('wombat);
       ⇒ #<lot :id 5>
create-lot('wombat);
```

⇒ **ERROR**

lot [Procedure]

Usage `lot(`*symbol*`);`

Description

This procedure returns the `lot` object for the lot with name *symbol*. If no such lot exists, `nil` is returned.

Examples

```
lot('cmos);
        ⇒ #<lot :id 1>
lot('foo);
        ⇒ nil
```

sublot-p [Procedure]

Usage `sublot-p(`*lot1, lot2*`);`

Description This procedure returns `t` if *lot1* is a sublot of *lot2*. *Lot11* is considered to be a sublot of *lot2* if every wafer in *lot1* is also in *lot2*.

lot-from-spec [Procedure]

Usage `lot-from-spec([name:` *symbol*`], {`*lot* | *wafer*`} {,`*lot* | ,*wafer*`}*);`

Description This procedure creates a new lot from a list of lot and wafers. If a `name` argument is passed, the name of the new lot is *symbol*.

Examples

```
lot-from-spec(name: 'foo, lot('cmos), lot('nwell));
        ⇒ #<lot :id 4>
sublot-p(lot('cmos), lot('foo));
        ⇒ t
```

current-lot [Procedure]

Usage `current-lot();`

Description This procedure returns a list of the names of the lots in the current lot.

Examples

```
with-lot '(cmos, nwell) do
  current-lot();
end;
        ⇒ (cmos, nwell)
```

deallocate-lot [Procedure]

Usage `deallocate-lot(`*lot*`);`

Description This procedure deallocates the specified lot.

Examples

```
l := lot('cmos);
        ⇒ #<lot :id 1>
deallocate-lot(l);
        ⇒ t
lot('cmos);
        ⇒ nil
```

lot-indexes [Procedure]

Usage `lot-indexes(`*lot*`);`

Description This procedure returns a list of all the indexes of the wafers in *lot*.

Examples

```
l := lot('cmos);
        ⇒ #<lot :id 1>
lot-indexes(l);
        ⇒ (1, 2, 3 ... 19, 20)
```

add-to-lot [Procedure]

Usage `add-to-lot(`*l*`, {,`*lot* `| ,`*wafer*`}+);`

Description `Add-to-lot` adds the supplied `wafer` or lot objects to the lot *l*.

Examples

```
lot-indexes(lot('cmos));
        ⇒ (1, 2, 3 ... 19, 20)
lot-indexes(lot('nwell));
        ⇒ (21, 25)
add-to-lot(lot('cmos),lot('nwell));
        ⇒ (1, 2, 3 ... 19, 20, 21, 25)
```

split-lot [Procedure]

Usage

```
split-lot(l, into: ({,lot-name}+));
split-lot(l, into: ({,lot-name, lot-size}+));
```

Description `Split-lot` splits the lot *l* into a number of sublots. The sublots can be specified in two ways. In the first case, a list of *lot-names* is passed, and the wafers in *l* are split and placed into the lots. The order in which wafers are split is as follows. The first wafer is put into the first lot. The second wafer is put into the second lot. If there are *j* lots, the *j*+1 wafer in *l* is put into the first lot, and so forth.

In the second case, a list of *lot-names* and *lot-sizes* are supplied. Wafers with the given names are created and the correct number of wafers is moved into them. It is an error if the sum of the *lot-size* arguments does not equal the size of *l*.

Examples

```
split-lot('cmos, into: '(high, med, low));
```

```
⇒ (#<lot :id 4>, #<lot :id 5> #<lot :id 6>)
```

`merge-lots` [Procedure]

Usage `merge-lots(`*lot1*`,` *lot2*`, name:` *symbol*`);`

Description This procedure creates a new lot named *symbol*, which includes of all of the wafers in *lot1* and *lot2*. *Symbol* can be the same as the name of *lot1* or *lot2*, in which case the new lot replaces the old lot.

`subtract-lots` [Procedure]

Usage `subtract-lots(`*lot1*`,` *lot2*`, name:` *symbol*`);`

Description This procedure creates a lot named *symbol* and fills it with wafers in *lot1* but not in *lot2*. *Symbol* can be the same as the name of *lot1* or *lot2*, in which case the new lot replaces the old lot.

`move-sublot` [Procedure]

Usage `move-sublot(`*l*`,` *from*`,` *to*`);`

Description This procedure removes the wafers in lot *l* from the lot *from* into the lot *to*.

`pick-wafer()` [Procedure]
`pick-test-wafer()` [Procedure]

Usage
```
pick-wafer();
pick-test-wafer();
```

Description These procedures return a wafer from the `current` lot. `Pick-wafer` returns any wafer from the current lot, whereas `pick-test-wafer` returns a wafer in `current` but not in `product`.

A.8 Equipment

Equipment is defined using **`defequipment`**. Database objects are created to represent equipment in a facility.

`defequipment` [Definition]

Usage **`defequipment`** *name* `((`*superclasses*`),` *attributes*`);`

Description `Defequipment` is used to define the interface to fabrication equipment for BPFL. *Name* is a symbol used to name the equipment. *Superclasses* is a list of names of an

equipment definition from which attributes will be inherited. *Attributes* is a list of attribute-value pairs for the equipment.

Examples

```
defequipment nanospec ((),
  recipes: (measure-oxide-thickness: (frame: nanospec)));
```

`allocate`	[Procedure]
`deallocate`	[Procedure]

Usage `allocate(`*equipment-spec*`);`
`deallocate(`*equip-obj*`);`

Description These procedures allocate and deallocate equipment. *Equipment-spec* is the name of an equipment class or a list of names. `Allocate` returns an object describing a piece of equipment that is a member of the equipment instances described by *equipment-spec*. If no suitable equipment is found, `nil` is returned. `Deallocate` takes an object describing a piece of equipment and deallocates it. It is an error to deallocate equipment that is not allocated for the current run.

The use of these procedures in user code is discouraged. `With-equipment` handles equipment allocation and deallocation, and guarantees that equipment is deallocated if a run encounters an error.

Examples

```
x := allocate('spinner);
       ⇒ #<equipment mti-spinner-1>
deallocate(x);
       ⇒ t
deallocate(x);
       ⇒ ERROR
```

A.9 Materials

Materials form a hierarchy. A partial listing of the material hierarchy is shown in Figure A-4. Each material class in the hierarchy has a unique *name* that is a BPFL symbol. When a material class is defined, a *parent* material is supplied. Materials also have *attributes*. When a material class is defined, the new material inherits attributes from its parent. Material definitions also include a list of *default-attributes* that can be used to override inherited values or to define new attributes. For example, the `silicon` material class is defined as

```
defmaterial si((), suprem-name: "SILICON");
defmaterial silicon((si), crystal: t);
defmaterial poly((si), simpl-name: "POLY",
```

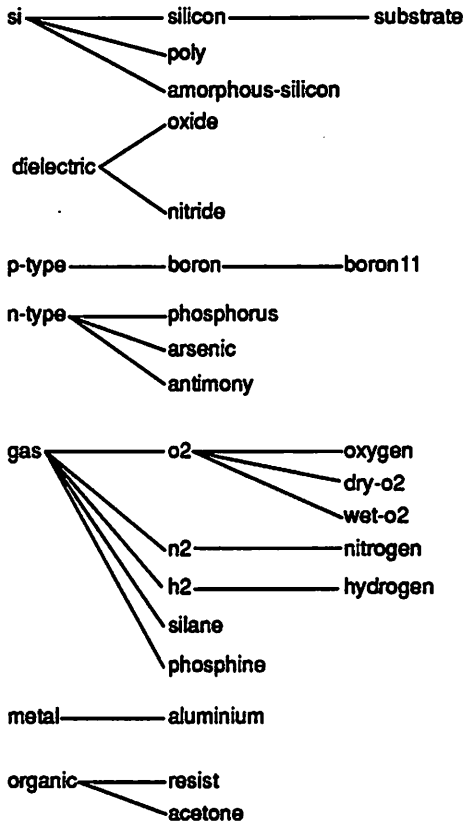

Figure A-4: BPFL material hierarchy.

```
suprem-name: "POLY");
```

In this case, the material has parent `si`, and one attribute `crystal`. Materials inherit default attributes from their parent, and attributes defined lower down the hierarchy override definitions higher up. For example, the `poly` material has `suprem-name` and `simpl-name` attributes with the value `"POLY."` The value of the `suprem-name` attribute defined for material `si` has been overridden.

Once material has been defined, the `material` procedure may be used to create a *material object.* Additional attributes may be attached to the material object, and the default values supplied in the material definition may be overridden For example, to create a material object of class `silicon`, the code

```
material(poly, grain-size: [{1 um}, {10 um}])
```

creates a poly material object with an additional attribute grain-size. The `material` procedure may be abbreviated to #m.

defmaterial [Definition]

Usage `defmaterial` *name* `((`*superclasses*`),`*attributes*`);`

Description `Defmaterial` is used to define a material for use in a process flow. *Name* is a symbol used to name the material. *Superclasses* is a list of the names of material from which this material inherits attributes. *Attributes* is a list of attribute names and values.

Examples

```
/* Material with no superclasses */
defmaterial silicon ((), atomic-weight: 28);
```

material [Procedure]
#m [Procedure]

Usage

```
material(name, attributes);
#m(name, attributes);
```

Description The `material` procedure creates a material object of class *name* and with the specified *attributes*. `Material` may be abbreviated to #m. The material object inherits any attributes present in the superclasses.

known-material-name [Procedure]

Usage `known-material-name(`*symbol*`);`

Description This procedure returns `t` if *symbol* is the name of a material class; otherwise `nil`.

Examples

```
known-material-name('foo);
        ⇒ nil
defmaterial((),foo);
        ⇒ #<material foo>
known-material-name('foo);
        ⇒ t
```

material-attr [Procedure]

Usage `material-attr(`*mat*`,` *attr*`[ ,`*default*`]);`

Description This procedure returns the value of the attribute *attr* for the material object *mat*. If *attr* is not an attribute of the material, *default* is returned. If *default* is not supplied, `nil` is returned.

Examples

```
defmaterial((si), poly, grain-size: [{5 um}, {10 um}]);
        ⇒ #<material poly>
material-attr(#m(poly), :grain-size);
        ⇒ [{5 um}, {10 um}]
```

```
material-attr(#m(poly), :foo, 0);
        ⇒ 0
```

`material-attrs` [Procedure]

Usage `material-attrs(`*mat*`);`

Description This procedure returns the names and values of all attributes of material *mat*. Attributes of the superclasses of *mat* are not included in the list.

`add-material-attrs` [Procedure]

Usage `add-material-attrs(`*mat* `{,`*attribute-name*`:` *attribute-value*`}+);`

Description This procedure creates a new material instance of the same class as *mat*, but with new attribute names and values as specified. If *mat* already has an attribute with the same name as an attribute to be added, the new value replaces the old value

---

`remove-material-attrs` [Procedure]

Usage `remove-material-attrs(`*mat* `{,:`*attribute-name*`}+);`

Description This procedure creates a new material instance of the same class as *mat*, but with the specified attributes removed.

`material-supers-list` [Procedure]

Usage `material-supers-list(`*mat*`);`

Description This procedure returns a list of all of the material classes higher up the material hierarchy than *mat*. The first element of list of the material class highest up the hierarchy.

`mtrl-class-match` [Procedure]

Usage `mtrl-class-match(`*m1*`,` *m2*`);`

Description This procedure returns `t` if *m1* and *m2* have the same material class, or if the class of one is the ancestor of the class of the other: otherwise `nil`.

Examples

```
mtrl-primary-match(#m(poly :annealed t), #m(poly));
        ⇒ t
mtrl-primary-match(#m(poly), #m(silicon));
        ⇒ t
mtrl-primary-match(#m(poly), #m(oxide));
        ⇒ nil
```

material-p [Procedure]

Usage material-p(*obj*);

Description Returns t if *obj* is a material object; otherwise nil.

Examples

```
material-p(#m(poly));
        ⇒ t
m := 56.4;
material-p(m);
        ⇒ nil
```

A.10 Masks, Layers, and Locations

A mask is defined using defmask. Masks have the following attributes: dark-field, type and edge. If dark-field is t, the mask is a *dark-field* mask and it shades that portion of the mask which is the logical inverse of the layer defining the mask. Type describes the type of mask, either chrome or emulsion. Edge describes the physical location of the edges of masks. For example, the clear-field emulsion METL mask is defined as

```
defmask METL(type: emulsion);
```

and the chrome dark-field contact-definition mask is defined as:

```
defmask CONT(dark-field: t, type: chrome);
```

The edge slot will ultimately be used to interface with a CAD database of mask data, but for now BPFL permits simple specification of the relationships between masks. This is achieved by specifying whether or not masks intersect (i.e., overlap). The default is that masks intersect, but masks may be declared to be disjoint (i.e., have no region in common) or one mask may be declared to be contained within another mask. For example, the CONT mask is contained within the METL mask, (i.e., every part of a wafer covered by CONT is also covered by METL). This fact may be indicated with the code

```
declare-contained-mask(CONT, METL);
```

Masks are declared to be disjoint with the procedure declare-disjoint-mask.

BPFL uses layers to describe regions of the wafer in terms of masks. For example, given a POLY mask and an ACTV mask:

```
defmask ACTV(type: emulsion);
defmask POLY(type: emulsion);
```

a suitable layer to express a region of the wafer that may be probed to determine active-area sheet-resistance is defined by

```
deflayer PROBE-ACTIVE(and(ACTV, not(POLY)));
```

That is, the active area not covered by polysilicon. Whenever a mask is defined using `defmask`, a layer with the same name as the mask is also defined.

Layers have four attributes: *name*, *position*, *edge* and *location*. The layer name is simply the symbol defining the layer. For example, the name of the layer defined in the above example is `PROBE-ACTIVE`. The layer position is the definition of the layer in terms of physical masks. For example, the position for the layer defined above is `and(ACTV, not(POLY))`. Layer edges are a logical expression of the edges of the layer in terms of the edges and areas of the masks making up the layer. For example, the edge attribute of the `PROBE-ACTIVE` layer is

```
OR(AND(ACTV,(EDGE POLY)),AND(NOT(POLY),(EDGE ACTV))
```

This expression means that the edges of the `PROBE-ACTIVE` layer are made up of the sum of two sets of edges. The first set is the edge of the `POLY` layer inside the `ACTV` layer (i.e., `AND(ACTV,(EDGE POLY))`. The second set is the edge of the `ACTV` layer outside the `POLY` layer (i.e., `AND(NOT(POLY),(EDGE ACTV))`.

The location slot is very similar to the position slot. The difference is that locations are dynamic and their values change depending upon the masks that have been used to expose the wafer. For example, consider the `PROBE-ACTIVE` layer definition, and suppose the wafer has been patterned with the `ACTV` mask but not yet patterned with the `POLY` mask. The `PROBE-ACTIVE` layer has location attribute

```
ACTV
```

because the wafer has not yet been patterned with the `POLY` mask, so the expressions involving POLY are ignored. That is, locations are created by evaluating positions with unused masks set to *don't-care* conditions.

Locations are used for two reasons. First, in PIF wafer-state representations they indicate where certain properties hold on a wafer. Second, they are used to indicate where measurements should be taken. For example, a measurement of gate oxide thickness could be expressed as:

```
measure-oxide-thickness(location: #1(PROBE-ACTIVE));
```

The `location` attribute of the `layer` argument is passed to the user to indicate where the measurement should be taken. The measured value is then stored in the wafer-state representation and the `location` is used to indicate where the measurement was taken.

BPFL *reduces* location expressions whenever possible. For example, since CONT is declared to be inside the METL layer, the location expression AND(METL, CONT) is reduced to the expression CONT, assuming that the wafer has been exposed to the METL and CONT masks. Likewise, OR(METL, CONT) is reduced to the expression METL. If layers are disjoint, as are NWELL and PWELL defined above, the expression AND(NWELL, PWELL) evaluates to `nil`. Layer reduction is used whenever possible to simplify location expressions. For example, if a location attribute evaluates to `nil`, then that location does not exist on the wafer. This property is used to delete sections of PIF descriptions that are completely removed during processing.

`defmask` [Definition]

Usage `defmask` *name* `[(`*attributes*`)];`

Description `Defmask` is used to define a mask for use in a process flow. *Name* is a symbol used to name the mask. *Attributes* is an optional list of attributes for the mask. It is expected that other attributes will be defined to support interfaces to CAD tools such as OCT [5].

Examples Definitions for a clear-field NWELL and POLY masks, and for dark-field CONT mask.

```
defmask NWELL (location: NWELL);
defmask POLY (location: POLY);
defmask CONT (location: not(CONT));
```

See Also `deflayer`

`known-mask-name` [Procedure]

Usage `known-mask-name(`*symbol*`);`

Description Returns `t` is *symbol* is the name of a mask; otherwise `nil`.

Examples

```
known-mask-name('POLY);
        ⇒ t
known-mask-name('foo);
        ⇒ nil
```

`mask` [Procedure]

Usage `mask(`*symbol*`);`

Description Returns the mask object with name *symbol*. It is an error if no such mask object exists.

Examples

```
mask('CONT)
        ⇒ #<mask CONT>
mask('foo)
```

⇒ **ERROR**

mask-name [Reader]

Usage `mask-name(`*m*`);`

Description Returns the name of the mask object *m*.

Examples

```
m := mask('POLY);
mask-name(m);
        ⇒ POLY
```

mask-attr [Procedure]

Usage `mask-attr(`*m*`, :`*attr-name*`);`

Description Returns the value of the attribute named *attr-name* from the mask object *m*.

Examples

```
mask-attr(mask('CONT), :dark-field);
        ⇒ t
mask-attr(mask('CONT), :type);
        ⇒ emulsion
```

deflayer [Definition]

Usage `deflayer` *name* `([`*location*`]);`

Description `Deflayer` is used to define a layer for use in a process flow. *Name* is a symbol used to name the layer. *Location* is an optional location specifier for the layer. This can be used to define the relationships between layers and masks.

Examples Define a single layer

```
deflayer NWELL;
deflayer CONT;
deflayer POLY;
```

Define a layer for probing contact cuts above the well.

```
deflayer CONT-PROBE and(CONT,NWELL);
```

See Also `defmask`

known-layer-name [Procedure]

Usage `known-layer-name(`*symbol*`);`

Description Returns `t` if *symbol* is the name of a layer; otherwise `nil`.

Examples

```
known-layer-name('POLY)
        ⇒ #<layer POLY>
known-layer-name('foo)
        ⇒ nil
```

layer	[Procedure]
#l	[Procedure]

Usage `layer(`*symbol*`);`

Description Returns the layer object with name *symbol*. It is an error if *symbol* is not the name of a known layer. `Layer` may be abbreviated as `#l`.

Examples

```
layer('NWELL);
        ⇒ #<layer NWELL>
#l('POLY);
        ⇒ #<layer POLY>
#l('foo);
        ⇒ ERROR
```

merge-layers	[Procedure]
intersect-layers	[Procedure]
invert-layer	[Procedure]

Usage

```
merge-layers(layer {, layer}+);
intersect-layers(layer {, layer}+);
invert-layer(layer);
```

Description `Merge-layers` returns a layer describing the union of the *layer* arguments.

`Intersect-layers` returns a layer describing the intersection of the *layer* arguments.

`Invert-layer` returns a layer describing all regions of the wafer outside *layer*.

Examples

```
merge-layers(#l(ACTV), #l(GATEOX));
        ⇒ #<layer ATCV>
intersect-layers(#l(ACTV), #l(GATEOX));
        ⇒ #<layer GATEOX>
invert-layer(#l(ACTV));
        ⇒ #<layer NOT(ACTV)>
```

layer-name	[Reader]
layer-definition	[Reader]
layer-location	[Reader]
layer-edges	[Reader]

Usage

```
layer-name(layer);
layer-definition(layer);
layer-location(layer);
layer-edges(layer);
```

Description Return the slots from the *layer* object

Examples

```
l := #l(GATEOX);
layer-name(l);
        ⇒ GATEOX
layer-location(l);
```

```
        ⇒ and(ACTV, not(POLY))
layer-edges(l);
        ⇒ #<edge OR(AND(ACTV,(EDGE POLY)),
                    AND(NOT(POLY),(EDGE ACTV)))>
```

References

[1] 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.

[2] G. L. Steele, *Common Lisp: The Language*, second edition, Digital Press, 1990.

[3] Franz Inc, *Common Lisp: The Reference*, Digital Press, 1989.

[4] S. G. Duvall, "An Interchange Format for Process and Device Simulation," *IEEE Trans. on CAD*, vol. 7, no. 7, pp 741–754, Jul. 1988.

[5] R. L. Spickelmeir, P. Moore, and A. R. Newton, *A Programmer's Guide to Oct.*, Electronics Research Lab. Memo, U.C. Berkeley.

[This page intentionally blank]

Appendix B

BPFL Implementation of Berkeley CMOS Process

B.1 Top-level flow (cmos-16.b)

```
require(cmos-lib, version: latest);

defflow cmos-16(implant-split: = t)
  "U.C. Berkeley Generic CMOS Process (Ver. 1.6 14-April-89)
  (2 um, N-well, single poly-Si, single metal)"
begin
  step ALLOCATE-WAFERS do
    let spec := bare-silicon-wafer(crystal-face: 100,
                                   resistivity: [{18 ohm-cm}, {22 ohm-cm}],
                                   quality: 'product, dope: 'p);
    begin
      allocate-lot(names: '(cmos, nwell, nch),
                   sizes: list(*product-lot-size*, 1, 1),
                   snapshot: spec);
    end;
    /* Wafers in the cmos lot are product wafers */
    lot('product) := lot('cmos);
    with-lot 'nwell do
        measure-bulk-resistivity(tag: "initial");
    end;
  end;

  with-lot 'cmos do
    step WELL-FORMATION do
      step INIT-OX do
        wet-oxidation(time: {11 min}, temperature: {1000 degC},
                      target-thickness: {1000 angstrom});
        pattern(mask-name: 'NWELL);
      end;
      step WELL-IMPLANT do
        with-lot '(cmos, nwell) do
          implant(species: #m(P), dose: {4.0e12 /cm^2},
                  energy: {150 keV});
          anneal-implant();
          etch-oxide(etchant: #m(BHF, dilution: 5/1));
          strip-resist();
          step DRIVE-IN do
            well-drive(temperature: {1150 degC}, time: {4 hr},
                       anneal-time: {5 hr});
            measure-oxide-thickness(location: #1(NWELL));
            measure-oxide-thickness(location: invert-layer(#1(NWELL)));
            with-lot 'nwell do
              etch-oxide(etchant: #m(BHF, dilution: 5/1));
              measure-sheet-resistance(location: #1(NWELL));
```

```
        end;
      end;
    end;
  end;
end;

step ACTIVE-AREA do
  with-lot '(cmos, nwell) do
    etch-oxide(etchant: #m(BHF, dilution: 5/1));
    constrain
      dry-oxidation(time: {28 min}, temperature: {950 degC});
      with-lot 'nwell do
        measure-oxide-thickness(tag: "LOCOS PAD");
        etch-oxide(etchant: #m(HF), dewet: t);
      end;
      nitridation(thickness: {1000 angstrom});

      when max-time-between('dry-oxidation,'nitridation,{20 min}) do
        etch-oxide(etchant: #m(HF), dewet: t);
        restart-body();
      end;
    end;
  end;
  pattern(mask-name: 'ACTV, will-double: t);
  etch-nitride(etchant: #m(nitride-etch-plasma, power: {50 W}));
  pattern(mask-name: 'PFIELD);
end;

step FIELD-IMPLANT do
  implant(species: #m(B11), dose: {1.0e13 /cm^2},
          energy: {100 keV});
  anneal-implant();
  strip-resist();
  wet-oxidation(temperature: {950 degC}, time: hms("3:20:00"),
                location: invert-layer(#1(ACTV)));
  with-lot '(cmos, nwell) do
    etch-nitride(etchant: #m(phosphoric-acid,
                             temperature: {145 degC}));
    dry-oxidation(temperature: {950 degC}, time: {28 min});
  end;
end;

step THRESHOLD-ADJUST do
  let species := #m(B11);
      energy := {30 keV};
      midrange := {1.2e12 /cm^2};
      delta := {0.1e12 /cm^2};
  begin
    if implant-split then
      split-lot('cmos, into: '(low, medium, high),
                       order: 'random);
      with-lot 'high do
        implant(species: species, dose: midrange + delta,
                energy: energy);
```

```
      end;
      with-lot 'medium do
        implant(species: species, dose: midrange, energy: energy);
      end;
      with-lot 'low do
        implant(species: species, dose: midrange - delta,
                energy: energy);
      end;
    else
      implant(species: species, dose: midrange, energy: energy);
    end;
    anneal-implant();
  end;
end;

step GATE-FORMATION do
  constrain
    /* Constraint body */
    with-lot '(cmos, nwell, nch) do
      dry-oxidation(target-thickness: {25 nm}, time: {40 min},
                    temperature: {950 degC});
    end;
    with-lot 'nwell do
      measure-oxide-thickness();
    end;
    with-lot 'nch do
      measure-oxide-thickness();
    end;
    with-lot 'cmos do
      deposit-doped-poly(thickness: {450 nm});
      /* deposit-doped-poly allocates a lot named poly
         with a poly control wafer in it */
    end;

    when max-time-between('dry-oxidation,
                          'deposit-doped-poly,
                          {20 min}) do
      etch-oxide(location: #1(ACTV), dewet: t);
      restart-body();
    end;
  end;
end;

step GATE-DEFINITION do
  pattern(mask-name: 'POLY);
  etch-poly(etchant: #m(poly-etch-plasma));
  strip-resist();
end;

step REOXIDATION do
  with-lot '(cmos, nwell, nch, poly) do
    wet-oxidation(time: {30 min}, temperature: {850 degC});
  end;
  with-lot 'nwell do
```

```
    measure-oxide-thickness();
  end;
  with-lot 'nch do
    measure-oxide-thickness();
  end;
  with-lot 'poly do
    measure-oxide-thickness();
  end;
end;

step NCHANNEL-S/D do
  pattern(mask-name: 'N-S/D);
  with-lot '(cmos, nch) do
    implant(species: #m(As), energy: {160 keV},
            dose: {5.0e15 /cm^2});
  end;
  strip-resist();
  with-lot '(cmos, nwell, nch) do
    anneal-implant(temperature: {950 degC}, time: hms("01:15:00"));
  end;
end;

step PCHANNEL-S/D do
  pattern(mask-name: 'P-S/D);
  with-lot '(cmos, nwell) do
    implant(species: #m(B11), dose: {2.0e15 /cm^2},
            energy: {50 keV});
  end;
  strip-resist();
  with-lot '(cmos, nwell, nch) do
    anneal-implant(temperature: {900 degC}, time: {15 min});
    measure-oxide-thickness(location: #1(P-S/D));
  end;
end;

step REFLOW-GLASS do
  with-lot 'cmos do
    deposit-psg(predoped-thickness: {200 nm},
                doped-thickness: {500 nm},
                postdoped-thickness: {100 nm},
                temperature: {450 degC});
  end;
  with-lot '(cmos, nwell, nch, psg) do
    densify-psg(time: {30 min}, temperature: {950 degC});
  end;
  with-lot 'nwell do
    measure-sheet-resistance(location: #1(TOP), tag: "P-S/D");
  end;
  with-lot 'nch do
    measure-sheet-resistance(location: #1(TOP), tag: "N-S/D");
  end;
  deallocate-lots('(nwell, nch, poly));
end;
```

```
step CONTACT do
  pattern(mask-name: 'CONT);
  etch-oxide(etchant: #m(oxide-etch-plasma), location: #1(CONT));
  rework-loop
    etch-oxide(etchant: #m(BHF, dilution: 10/1), location: #1(CONT),
               thickness: {10 nm});
    rework-test contact-probe(location: #1(CONTACT-TEST));
  end;
  strip-resist(etchant: #m(oxygen-plasma));
end;

step BACK-SIDE-ETCH do
  spin-soft-bake();
  hard-bake-resist(double-photo: t);
  spin-soft-bake();
  constrain
    hard-bake-resist();
    etch-oxide(etchant: #m(BHF), dewet: t, location: #1(BACK-SIDE));
    etch-poly(etchant: #m(phosphoric-acid),
              location: #1(BACK-SIDE));
    etch-oxide(etchant: #m(BHF), dewet: t, location: #1(BACK-SIDE));
    when max-time-between('hard-bake-resist, 'etch-oxide,
                          {30 min}) do

      restart-body();
    end;
  end;
  strip-resist(etchant: #m(oxygen-plasma));
end;

step METALLIZE do
  deposit-al(thickness: {600 nm});
  pattern(mask-name: 'METL, resist: #m(wx-235));
  etch-al();
  strip-resist(etchant: #m(oxygen-plasma));
  test("Metal integrity tests");
end;

step SINTER do
  fast-sinter();
end;

step PRE-PASSIVATION-TEST do
  test("2 um NMOS & PMOS devices and capacitors");
end;

step PASSIVATION do
  deposit-pecvd-oxide(thickness: [{700 nm}, {800 nm}]);
  deallocate-lots('pecvd);
  pattern(mask-name: 'PASSIV);
  etch-oxide(etchant: #m(oxide-etch-plasma));
  strip-resist(etchant: #m(oxygen-plasma));
end;
```

```
    step FINAL-TEST do
      test("Final functional test");
    end;
  end;
end;


defflow test (description)
  "Puts up a form requesting device test. Description is
   descriptive text indicating what is to be tested"
let result := nil;
begin
  /* There is no form to display in this case because test
     devices and results depend on the circuit being fabricated.
     Use frame type general to display rudimentary information and
     permit the user to enter comments */
  result := user-dialog('general, heading: "cmos probe test",
                        description: description);
  wip-log('general, result);
end;
```

B.2 Outline of CMOS Library (cmos-lib.b)

The following is a list of the top-level declarations in the `cmos-lib` library, and a brief description of the arguments and actions of functions called by the `cmos-16` process flow.

```
require(material, version: latest);
require(pif-wafer, version: latest);
require(equipment, version: latest);
require(physical-constants, version: latest);
require(litho, version: latest);
require(ucb-std, version: latest);


defflow wet-oxidation(time:, temperature: = {900 degC},
                      pre-ox-time: = {5 min}, post-ox-time: = {5 min},
                      anneal-time: = {20 min}, target-thickness:,
                      tag:)
  "Cleans wafers and furnace, performs wet oxidation and measures
   oxide thickness on a test wafer in the current lot"
begin
  ...
end;


defflow dry-oxidation(time:, temperature: = {900 degC},
                      anneal-time: = {20 min}, target-thickness:,
                      tag:)
  "Cleans wafers and furnace, performs dry oxidation and measures
   oxide thickness on a test wafer in the current lot"
begin
  ...
end;


defflow nitridation(thickness:, temperature: = {800 degC})
  "Cleans wafers, grows nitride and measures nitride thickness
   on a test wafer in the current lot"
begin
  ...
end;


defflow implant(species:, dose:, energy:, tag:)
  "Implants wafers and performs anneal"
begin
  ...
end;
```

```
defflow n2-anneal(time:, temperature:)
  "Nitrogen anneal for specified time and temperature"
begin
  ...
end;


defflow anneal-implant(time: = {30 min}, temperature: = {950 degC})
  "Calls n2-anneal with specified time and temperature suitable
   for standard implant anneals"
begin
  ...
end;


defflow well-drive(time:, temperature: = {1150 degC},
                   anneal-time:, target-thickness:, tag:)
  "Performs wet oxidation to drive in well"
begin
  ...
end;


defflow deposit-doped-poly(time:, temperature: = {650 degC}, thickness:)
  "Doped poly deposition. Performs deposition at specified time
   and temperature or at time required to achieve specified thickness
   based on deposition rate from equipment log.
   Allocates a lot poly with one control wafer."
begin
  ...
end;


defflow deposit-undoped-poly(time:, temperature: = {650 degC},
                             thickness:)
  "Undoped poly deposition. Performs deposition at specified time
   and temperature or at time required to achieve specified thickness
   based on deposition rate from equipment log.
   Allocates a lot poly with one control wafer."
begin
  ...
end;


defflow deposit-pecvd-oxide(time:, temperature: = {250 degC},
                            thickness:)
  "Pecvd oxide deposition. Performs deposition at specified time
   and temperature or a time required to achieve specified thickness
   based on deposition rate from equipment log."
begin
  ...
end;
```

```
defflow deposit-al(thickness:)
  "Sputter Al of the desired thickness"
begin
  ...
end;


defflow deposit-psg(temperature: = {450 degC},
                    predoped-thickness:, predoped-time:,
                    doped-thickness:, doped-time:,
                    postdoped-thickness:, postdoped-time:)
  "Deposits PSG in three layers.
   First an undoped layer of thickness predoped-thickness,
   second a doped layer of thickness doped-thickness,
   third an undoped layer of thickness postdoped-thickness.

   The arguments predoped-time, doped-time and postdoped-time
   can be used to specify absolute deposition times"
begin
  ...
end;


defflow densify-psg(time:, temperature:)
  "Wet oxidation to densify glass"
begin
  ...
end;


defflow fast-sinter(time: = {20 min}, temperature: = {400 degC})
  "Sinter wafers with no rampup"
begin
  ...
end;


/* The following functions each measure a physical quantity.
   The wafer-state is queried to determine expected values, if
   any. A test wafer is used if one exists in the current lot,
   otherwise a product wafer is used */

defflow measure-oxide-thickness (tag:, location:)
begin
  ...
end;
```

```
defflow measure-bulk-resistivity(tag:, location:)
begin
  ...
end;


defflow measure-sheet-resistance(tag:, location:)
begin
  ...
end;


defflow measure-poly-thickness(tag:, location:)
begin
  ...
end;


defflow measure-nitride-thickness(tag:, location:)
begin
  ...
end;

/* The following functions each etch a particular material on
   the wafers in the current lot. Each takes at least the following
   arguments:
      etchant, thickness, overetch, location
   Etchant is a material used for the etching (e.g.,
     #m(nitride-etch-plasma) or #m(phosphoric-acid) to etch
     nitride.
   Thickness is the thickness of material to etch. It is
     ordinarily NOT used because snapshots are queried to determine
     the thickest exposed layer of material and that thickness is
     etched.
   Overetch is added to thickness to be etched. Normally expressed
     as a percentage. Etch functions usually have a default
     value for overetch  (e.g., {10 %}).
   Location is a layer indicating which region of the wafer
     the user should use for endpoint detection. This argument
     is only used if the thickness argument is supplied. If the
     etchant is a liquid and the back-side of the wafers
     have thickness equal to the thickest exposed layer of material,
     location takes the value #l(BACK-SIDE) */
```

```
defflow etch-oxide(etchant:, thickness:, overetch: = {10 %}, location:,
                   dewet:)
  "Dewet is used to force complete removal of exposed oxide
   The following sanity checks are performed:
    Warning generated if wet etches occur after
     poly is deposited.
    Error generated if wafers with exposed metal
     are wet etched."
begin
  ...
end;


defflow etch-nitride(etchant: = #m(phosphoric-acid), thickness:,
                     overetch: = {5 %}, location:)
begin
  ...
end;


defflow etch-poly(etchant: = #m(poly-etch-plasma), thickness:,
                  overetch: = {5 %}, location:)
begin
  ...
end;


defflow etch-al(etchant: = #m(al-wet-etcher), thickness:,
                overetch: = {5 %}, location:)
begin
  ...
end;
```

B.3 Litho Library (litho.b)

```
defmaterial developer ((organic));
defmaterial kodak932 ((developer));
defmaterial mif ((developer));

defmaterial resist ((organic), exposed: nil);
defmaterial kodak-820 ((resist),
  negative: nil,
  spin-speed: {4600 rpm},
  prebake-temp: [{95 degC}, {100 degC}],
  prebake-time: {30 s},
  exposure: {130 mJ/cm^2},
  developer: #m(kodak932, concentration: {50 %},
                          temperature: {22 degC},
                          time: [{30 s}, {60 s}]),
  hard-bake: (temp: {120 degC}, time: {20 min},
              double-photo: (temp: {150 degC}, time: {30 min})));


defmaterial wx-235 ((resist),
  negative: nil,
  spin-speed: {4000 rpm},
  prebake-temp: {110 degC},
  prebake-time: {30 s},
  exposure: {140 mJ/cm^2},
  developer: #m(mif, concentration: {33 %},
                     temperature: {90 degC},
                     time: {90 s}),
  hard-bake: (temp: {100 degC}, time: {20 min},
              double-photo: (temp: {100 degC}, time: {30 min})));


defequipment developer ((),
  recipes: (develop: (frame: develop-resist),
            strip: (frame: strip-resist)));


defequipment wafer-track ((),
  recipes: (spin-soft-bake: (frame: spin-soft-bake)));


defequipment stepper ((),
  recipes: (expose: (frame: expose-resist)));


defflow pattern(mask-name:, will-double:, resist: = *default-resist*)
 "Basic photolithography - coat, expose, develop, descum, bake"
let double-photo := find-surface-segments-in-lot(material: #m(resist));
begin
  step PATTERN do
    rework-loop
      constrain
        spin-soft-bake(resist: resist);
        expose-resist(mask-name: mask-name);
```

```
        develop-resist();
        when (max-time-between('spin-soft-bake, 'expose-resist,
                               {2 day}) or
              max-time-between('expose-resist, 'develop-resist,
                               {1 hour})) do
          halt-run("time-constraint-violation in pattern");
        end;
      end; /* constrain */
      rework-test inspect-resist();
      retry-count 5;
      rework-prefix if not(double-photo) then
                      strip-resist();
                    end;
    end; /* rework */
    descum-resist();
    hard-bake-resist(double-photo: (double-photo or will-double));
  end; /* step */
end;


defflow spin-soft-bake(resist: = *default-resist*)
 "dehydrate, hmds treat and spin resist onto wafers"
let last-dehyd-time := dehydrate-wafers();
begin
  constrain
    deposit-hmds();
    deposit-resist(resist: resist);
    when max-time-between(last-dehyd-time,'deposit-resist,{30 min}) do
      last-dehyd-time := dehydrate-wafers();
      restart-body();
    end;
    when max-time-between('deposit-hmds,'deposit-resist,{10 min}) do
      restart-body();
    end;
  end; /* constrain */
end;


defflow dehydrate-wafers()
  "Dehydrate wafers if necessary and return the dehydration time."
let segments := find-segments-in-lot(material: #m(substrate));
begin
  view fabrication do
    if (min(segment-material-attribute-in-lot(segments,
                                              :dehydration-time))
      + {30 min}) < current-time() then
      with-equipment o of-type 'oven do
        run-recipe(o, 'dehydrate-wafers);
        segment-material-attribute-in-lot(segments,
             :dehydration-time) := last-equip-time();
      end;
    end;
  end;
```

```
  min(segment-material-attribute-in-lot(segments, :dehydration-time));
end;


defflow deposit-hmds()
  "Coat the wafers with hmds"
begin
  user-dialog('hmds-coat);
end;


defflow deposit-resist(resist: = *default-resist*)
  "Coat the wafers with resist"
begin
  view fabrication do
    with-equipment track of-type 'wafer-track do
      run-recipe(track,'spin-soft-bake,
                 resist-name: material-name(resist));
    end;
  end;
  deposit-material-in-lot(resist);
end;


defflow expose-resist(mask-name:)
  "Expose wafers "
let layer := find-layer(mask-name); /*Layer corresponding to the mask */
    exposure-location := intersect-layers(top-side(),
                                  invert-layer(layer));
    old-segments := nil;
    new-segments := nil;
begin
  viewcase
    when fabrication do
      with-equipment stepper of-type 'stepper do
        run-recipe(stepper, 'expose, mask-name: mask-name);
      end;
    end;
  end;
  old-segments :=
    find-segments-in-lot(material: #m(resist, exposed: nil));
  new-segments :=
    split-segments-in-lot(old-segments, location: exposure-location);
  segment-material-attribute-in-lot(new-segments, :exposed) := t;
end;


defflow develop-resist()
  "Develop resist in lot"
begin
  viewcase
    when simpl do
      simpl-op("DEVL","ERST");
    end;
```

```
    when fabrication do
      with-equipment d of-type 'developer do
        run-recipe(d, 'develop-resist,
                   resist-name: material-name(resist-in-lot()));
      end;
    end;
  end;
  etch-material-in-lot(#m(resist, negative: nil, exposed: t), t);
  etch-material-in-lot(#m(resist, negative: t, exposed: nil), t);
end;


defflow hard-bake-resist(double-photo:, time:, temperature:)
  "Hard bakes resist. Uses parameters in resist definition unless
   time and temperature arguments are supplied"
begin
  let resist := resist-in-lot();
      bake-attr := material-attr(resist, :hard-bake);
      bake-time := nil;
      bake-temp := nil;
      segments := find-segments-in-lot(material: #m(resist));
  begin
    if double-photo then
      bake-attr := getf(bake-attr, :double-photo);
    end;
    if (was-supplied(time) and was-supplied(temperature)) then
      /* Arguments are used, so must check type */
      assert(unit-with-dimensions-p(time,"s"),
             "Must be a unit quantity with dimensions of time",
             time);
      assert(unit-with-dimensions-p(temperature,"K"),
             "Must be a unit quantity with dimensions of temperature",
             temperature);
      bake-time := time;
      bake-temp := temperature;
    else
      bake-time := getf(bake-attr, :time);
      bake-temp := getf(bake-attr, :temp);
    end;
    view fabrication do
      user-dialog('hard-bake, time: bake-time, temp: bake-temp);
    end;
    segment-material-attribute-in-lot(segments, :last-bake-time) :=
      current-time();
  end;
end;


defflow inspect-resist()
  "Inspect each wafer and put wafers to be reworked into the rework lot
   and wafers to be scrapped into the scrap lot."
begin
  view fabrication do
    with-equipment scope of-type 'microscope do
```

```
      let results := user-dialog(name: 'inspect-resist,
                                 equipment: scope);
      begin
        wip-log('Resist-Inspect, results);
        move-sublot(getf(results, :rework), 'current, 'rework);
        move-sublot(getf(results, :scrap), 'current, 'scrap);
        if lot('rework) then
          raise-exception('rework);
        end;
      end;
    end;
  end;
end;


defflow strip-resist(etchant: = #m(acetone))
  "Removes resist from wafers in the current lot"
let equip-type := 'developer;
begin
  if etchant = #m(oxygen-plasma) then
    equip-type := 'resist-plasma-etcher
  end;
  view fabrication do
    with-equipment s of-type equip-type do
      run-recipe(s,'strip, etchant: etchant);
    end;
  end;
  etch-material-in-lot(#m(resist),t);
end;


defflow resist-in-lot()
  "Utility routine that returns the resist on the wafers in
   the current lot. Signals an error if more than one type of
   resist is present"
let last-resist := nil;
    ss := lot-snapshots();
begin
  for-each seg in find-segments-in-lot(material: #m(resist)) do
    this-resist := pif-attr-val(seg, :material, ss);
    if last-resist then
      if material-name(last-resist) != material-name(this-resist) then
        halt-run("Two different types of resist on wafers: ~a and ~a",
                 last-resist, this-resist);
      end;
    end;
    last-resist := this-resist;
  end;
  last-resist;
end;
```

Appendix C
ABF Functions

C.1 Introduction

This document describes the functions for accessing compound BPFL data types from ABF, and for manipulating `dialog` and `WIP-log` records. The first section outlines the functions for accessing frame arguments. The second section describes the function used to check data entered by a user, and the third section discusses functions available from within `user-dialog` frames to append attributes to the WIP-log.

C.2 Argument and object accessor functions

The WIP interpreter passes frame arguments to the UI process via the database. Each argument has a *name* and a *value*. For example, the `user-dialog` call

```
user-dialog('sonogage, wafer-id: 1,
                       nominal: [{1 ohm-cm}, {10 ohm-cm}]);
```

calls the `sonogage` frame with two arguments named `wafer-id` and `nominal`. The values of the arguments are `1` and `[{1 ohm-cm}, {10 ohm-cm}]` respectively.

The WIP interpreter automatically passes a `lot` argument to every `user-dialog` frame. The value of this argument is a list of the names of the lots making up the `current` lot. For example, the code

```
with-lot '(cmos, nwell) do
   user-dialog('spin-soft-bake);
end;
```

calls the `spin-soft-bake` frame with one argument named `lot` with a value of `(cmos, nwell)`.

All of the functions discussed in this section can be used in `user-dialog` and `WIP-log` frames. Frame arguments may be read and attributes may be appended to the WIP-log in `user-dialog` frames. WIP-log attributes may be read in `WIP-log` frames.

As an example of the available accessor functions, suppose a frame is called with the following arguments:

Accessor function	Description
`arg_supplied(`*name*`)`	Returns 1 if the argument named *name* has been passed to the frame; otherwise 0.
`arg_value(`*name*`)`	Returns a string containing the print representation of the argument *name*. If the argument has not been passed to the frame, returns `null`.
`arg_pointer(`*name*`)`	Returns a pointer to the object describing the argument *name*. If the argument has not been passed to the frame, returns `null`.

Table A-1: Argument accessor functions.

```
user-dialog(frame, recipe-name: "SWETOXB",
                   step-number: 5,
                   target-thickness: [{85 nm}, {90 nm}],
                   temperature: {1000 degC},
                   equipment-list: (tylan1, tylan2));
```

All functions discussed below will be assumed to execute in the ABF code for the frame called by the `user-dialog`.

The `arg_supplied` function is used to determine whether or not an argument has been passed to a frame. `Arg_supplied` takes a single string argument containing the name of the desired frame argument, and returns 1 if the argument exists and 0 if it does not. For example,

```
arg_supplied('recipe-name');
⇒ 1
arg_supplied('foo');
⇒ 0
```

The `arg_value` function returns a string containing the value of an argument. It returns a null string if no such argument exists.

```
arg_value('equipment-list');
⇒ '(tylan1, tylan2)'
arg_value('foo');
⇒ null
```

The argument accessor functions are summarized in Table A-1.

ABF programs do not have functions to manipulate compound BPFL data types (see Appendix A, section A.3.2), so argument values are manipulated by object accessor functions. The `arg_pointer` function returns a pointer to an object describing an argument. This pointer can be assigned to an ABF `integer` variable which can be passed to object accessor functions. For example, the `integer_obj` function returns the integer value of the object pointed to by its argument:

```
/* Assume x is an integer variable */
x := arg_pointer('step-number');
integer_obj(x);
⇒ 5
```

Similarly `string_obj` and `float_obj` access string and floating-point numbers respectively. If an incorrect accessor function is used (e.g., `string_obj` is used to access the value of an integer object), an error message is displayed and the accessor function returns `null`. The `object_type` function returns a string that describes the type of an object:

```
object_type(arg_pointer('step-number'));
⇒ 'integer'
object_type(arg_pointer('equipment-list'));
⇒ 'list'
```

There is one situation in which an incorrect accessor function will yield a correct result: it is legal to use `float_obj` to access the value of an integer object. For example,

```
float_obj(arg_pointer('step-number'))
⇒ 5.0
```

Accessors are provided to select parts of a compound object. These functions return pointers to the selected part of the object. For example, intervals have two accessors, `interval_left` and `interval_right`:

```
x := arg_pointer('target-thickness');
object_type(interval_left(x));
⇒ 'unit'
```

Units also have two accessor functions, `unit_magnitude` and `unit_dimension`.

```
x := interval_left(arg_pointer('target-thickness'));
integer_obj(unit_magnitude(x));
⇒ 85
string_obj(unit_dimension(x));
⇒ 'nm'
```

Complex numbers have `complex_real` and `complex_imag` accessor functions for accessing the real and imaginary components of a complex number respectively.

Lists can be of any depth and length. The `list_length` function returns the length of a list object passed to it. For example,

```
list_length(arg_pointer('equipment-list'));
⇒ 2
```

Elements in lists are accessed by the `list_elt` function, which returns a pointer to the object describing the element. The function call `list_elt(x,n)` where `x` is a pointer to a list object and `n` is an integer less than or equal to the length of the list pointed to by `x` accesses the n^{th} element of

Accessor function	Description
`object_type(`*obj* `)`	Returns a string containing the type of object pointed to by *obj*. The possible return values are: 'integer', 'float', 'complex', 'interval', 'unit' and 'list'. `Null` is returned if *obj* does not point to an object.
`object_printrep(`*obj* `)`	Returns a string containing a print representation for the object pointed to by *obj*. `Null` is returned if *obj* does not point to an object.

`integer_obj(`*obj* `)`
`float_obj(`*obj* `)`
`string_obj(`*obj* `)`
	Returns an integer, float or string value for the object pointed to by *obj*. If the object has the wrong type, returns `null` except under the following condition: if *obj* points to an integer object, `float_obj` returns a floating point number with the same value as the integer.

`interval_left(`*obj* `)` `interval_right(`*obj* `)`	If *obj* points to an interval object, returns a pointer to the left or right part of the interval respectively. If *obj* does not point to an interval, returns `null`.
`unit_magnitude(`*obj* `)` `unit_dimension(`*obj* `)`	If *obj* points to a unit object, returns a pointer to the magnitude or dimension of the interval respectively. If *obj* does not point to a unit, returns `null`.
`list_length(`*obj* `)` `list_elt(`*obj, i*`)`	If *obj* points to a list object, `list_length` returns an integer value for the length of the list. List_elt returns the i^{th} element of the list object pointed to by *obj*. If *obj* does not point to a list object or if *i* is greater than the length of the list, `null` is returned.

Table A-2: Object accessor functions.

the list. For example,

```
string_obj(list_elt(arg_pointer('equipment-list'),1));
⇒ 'tylan1'
```

Any object can be converted to a string with the `object_printrep` function. This function returns a string containing a print representation of an object. This is useful if an object is to be displayed in an ABF field. For example,

```
x := interval_left(arg_pointer('target-thickness'));
object_printrep(x);
⇒ '{85 nm}'
```

The object accessor functions are listed in Table A-2.

C.3 Checking User Input

Since ABF does not directly support compound data types, all compound data types are entered into string fields. Therefore, a mechanism must be provided to perform syntax checks on values entered by users. For example, if a user types the string `"18.2.2 ohm-cm"` into a field, it must be possible for the program to report an error and force the user to correct the entry.

`Check_format` is the function used to check the syntax of values. It also performs range checking on interval types and dimension checking of unit types. It takes a variable number of arguments. The first argument is always the value to be checked and the second argument is a *format string* that specifies the type of quantity to be checked. For example, the function call:

Format String	Parses
{}	Any unit. (e.g., {18 ohm-cm}, {(0.1, 0.2) ohm})
{integer}	Integer unit.
{float}	Floating point unit.
{complex}	Complex unit.
[]	Any interval.
[integer]	Integer interval.
[float]	Floating point interval.
[{}]	Unit interval.
[{integer}]	Integer unit interval.
[{float}]	Floating point unit interval.

Table A-3: `Check_format` format strings.

```
check_format(value,"[]");
```

checks the syntax of the string stored in the `value` variable. It returns 0 if the string is correctly formatted to be a unit type and a negative integer if the string is not correctly formatted. For example:

```
value := "15.0 ohm-cm";
check_format(value,"[]");
⇒ 0
check_format("82.2.2 ohm-cm", "[]");
⇒ -1
```

The acceptable format strings and the data types they parse are listed in Table A-3. The user does not have to type the unit and interval delimiters (i.e., `{}` and `[]`). For example, both the string `"{1.2 ohm-cm}"` and the string `"1.2 ohm-cm"` are acceptable if parsed with a format string of `"{float}"` or `"{complex}"`. Furthermore, intervals of unit type can be entered by using an implicit dimension of the left-hand side of the interval. For example, if the user wishes to enter a string denoting an interval of 1 micron to 22.5 microns, any of the following strings is acceptable:

```
"[{1 um} {22.5 um}"
"{1 um} {22.5 um}"
"1 um 22.5 um"
"1-22.5 um"
```

Further arguments are passed to `check_format` to perform dimension checking of unit types and range checking of interval types. Dimension checking is accomplished by supplying a third argument that is a string containing the desired dimensions. For example:

```
check_format(f,"{float}","ohm-cm");
```

checks the string stored in the `f` variable and returns 0 if it is a floating point unit type with dimensions of ohm-cm. If the value of `f` is `"18.2 ohm-cm"`, `check_format` returns 0. If the value of `f` is `"18.2 m"`, `check_format` returns a negative integer. Dimensions are only checked for

Return Value	Description
0	No error.
-1	Malformed input string.
-2	Malformed dimension.
-3	Invalid number type.
-4	Inconsistent units.
-5	Out of range.

Table A-4: `Check_format` return values.

dimensional consistency, so if `f` has the value `"18.2 V-m/kA"` `check_format` returns 0 because `ohm-cm` and `V-m/kA` are consistent dimensions.

Range checking is accomplished by providing two extra arguments to `check_format`. For example:

```
check_format(f,"(float)","1 ohm-cm","100 ohm-cm");
```

checks the value of `f` and returns 0 if it is a floating point unit with dimensions of `ohm-cm` and a value between 1–100 ohm-cm. If the value of `f` is one of the following:

```
"1.2 ohm-cm"
"56.7 ohm-cm"
"99 ohm-cm"
```

`check_format` returns 0. On the other hand, if `f` has the value `"0.1 ohm-cm"` or `"9 V-m/kA"`, `check_format` returns a negative value.

`Check_format` does not alter the string it parses. If the string fails a parse, `check_format` always displays an error message indicating the reason for the error and returns a negative integer. The possible return values for `check_format` and their descriptions are shown in Table A-4. The values returned by `check_format` are normally used by ABF programs to force the user to enter a correct value. In other words, ABF programs are usually written to force the user to correct errors in entered values before moving on to the next operation. This is not always desirable. For example, suppose a user is measuring some quantity (e.g., bulk resistivity) and types that value into a field with `check_format` used to check the range of the entered value. If the user enters a value outside the acceptable range (i.e., `check_format` returns -5), it is often better to simply warn the user that the value is out of range and accept the value, because it is possible that the entered value has been correctly typed by the user but that the range specified in the check is too narrow.

C.4 Appending Attributes to the WIP log

All arguments passed to a user-dialog frame are written to the WIP-log record for the user-dialog frame and become attributes in the WIP-log entry. In addition, users may append new attributes to the WIP-log. For example, the nanospec frame is used to enter measurements of film thickness, and a `measurements` attribute is added to the WIP-log for the `nanospec`.

Attributes are added to the WIP-log with the `create_log_attr` function. This function takes a single argument that is the name of the desired attribute to be added. For example, the following function call adds a `measurements` attribute to the WIP-log:

```
create_log_attr('measurements');
```

An error message is generated if an attribute with the same name already exists.

Attributes created with `create_log_attr` have no value. Attribute values can be set by using the `set_attr_value` function. This function takes three arguments:

1. the name of the attribute whose value is to be set,
2. the value to be assigned to the attribute, and
3. the value type.

For example, the following function call sets the value of the `measurement` attribute to a unit value of `{18.2 ohm-cm}`:

```
set_attr_value('measurements','18.2 ohm-cm', '{}');
```

The third argument is used by `set_attr_value` to parse the value passed to it and make sure it has the correct syntax.

Lists can also be assigned to attributes. The current implementation does not permit lists to be elements of lists, so an attribute list is a simple list of string, integer, floating point, complex, unit or interval values. Values are appended to an attribute list by using the `append_attr_value` function. This function takes the same arguments as `set_attr_value`. Values appear in the list in the same order in which they are appended using `append_attr_value`. For example, the following sequence of calls creates a `measurements` attribute with value of `({18.2 ohm-cm} {21.0 ohm-cm} {23.9 ohm-cm})`:

```
create_log_attr('measurements');
append_attr_value('measurements','18.2 ohm-cm', '{}');
append_attr_value('measurements','21.0 ohm-cm', '{}');
append_attr_value('measurements','23.9 ohm-cm', '{}');
```

Normally list attributes are created by using the ABF `unloadtable` statement to append values from a table field into an attribute.

Appendix D
Database Definition

D.1 Introduction

This document contains Common Lisp code to create the database used by the WIP system. Ingres does not allow hyphens in table or field names, so an underscore (_) is used instead. The WIP interpreter accesses the database through CLING/SQL [54].

D.2 Definition

```
;; The following defvar creates constants for field lengths and writes the .h
;; file used by ABF programs.
;;
(defvar *DB-BUFFER-LENGTH*
    (let ((m 0))
      (with-open-file (out "wip-db.h" :direction :output :if-exists :supersede)
        (dolist (dd '((*DB-FRAME-TYPE-LENGTH* 40)
                      (*DB-FRAME-SLOTS-LENGTH* 1500)
                      (*DB-BINDINGS-LENGTH* 1500)
                      (*DB-FUNCTION-NAME-LENGTH* 100)
                      (*DB-PACKAGE-NAME-LENGTH* 100)
                      (*DB-FUNCTION-CODE-LENGTH* 1500)
                      (*DB-PACKAGE-USE-LENGTH* 100)
                      (*DB-STEP-PATH-LENGTH* 200)
                      (*DB-LOG-ARG-LENGTH* 500)
                      (*DB-LOG-NAME-LENGTH* 40)
                      (*DB-LOG-TAG-LENGTH* 40)
                      (*DB-LOG-PROC-LENGTH* 40)
                      (*DB-WAFER-SPEC-LENGTH* 100)
                      (*DB-WAFER-SCRIBE-LENGTH* 10)
                      (*DB-REWORK-STACK-LENGTH* 200)
                      (*DB-EQUIP-NAME-LENGTH* 30)
                      (*DB-EQUIP-PROG-NAME-LENGTH* 30)
                      (*DB-EQUIP-PROG-DESCRIP-LENGTH* 60)
                      (*DB-REWORK-FRAME-ID-LENGTH* 20)
                      (*DB-COMMENT-LENGTH* 60)
                      (*DB-PF-ARG-NAME-LENGTH* 20)
                      (*DB-PF-ARG-DEF-LENGTH* 20)
                      (*DB-PF-TYPE-LENGTH* 10)
                      (*DB-PF-NAME-LENGTH* 20)
                      (*DB-PF-VERSION-LENGTH* 20)
                      (*DB-PF-COMMENT-LENGTH* 80)
                      (*DB-PF-FILE-NAME-LENGTH* 30)
                      (*DB-PF-STATE-LENGTH* 20)
                      (*DB-RUN-NAME-LENGTH* 20)
                      (*DB-RUN-STATUS-LENGTH* 10)
                      (*DB-RUN-STEP-LENGTH* 10)
                      (*DB-RUN-LOCK-LENGTH* 10)
                      (*DB-USER-NAME-LENGTH* 20)
```

```
                    (*DB-MAIL-ADDRESS-LENGTH* 40)
                    (*DB-UDEFAULT-LENGTH* 20)
                    (*DB-MASK-SET-NAME-LENGTH* 20)
                    (*DB-MASK-NAME-LENGTH* 20)
                    (*DB-MASK-TYPE-LENGTH* 20)
                    (*DB-MASK-LOCATION-LENGTH* 20)
                    (*DB-MATERIAL-PRIMARY-LENGTH* 20)
                    (*DB-MATERIAL-REST-LENGTH* 100)
                    (*DB-PIF-TYPE-LENGTH* 20)
                    (*DB-PIF-ATTR-NAME-LENGTH* 20)
                    (*DB-PIF-ATTR-VALUE-LENGTH* 80)
                    (*DB-SNAPSHOT-SEGMENTS-LENGTH* 80)
                    (*DB-SNAPSHOT-HASH-LENGTH* 80)
                    (*DB-LOT-NAME-LENGTH* 20)
                    (*DB-LOT-BITS-LENGTH* 10)
                    (*DB-WAFER-SCRIBE-LENGTH* 20)
                    (*DB-LAYER-NAME-LENGTH* 20)
                    (*DB-LAYER-DEFN-LENGTH* 40)
                    (*DB-LAYER-CACHE-LENGTH* 20)
                    )
                m)
        (eval (cons 'defconstant dd))
        (format out "#define ~a (~a)~%"
                (convert-to-c-name (first dd))
                (second dd))
        (setf m (max m (second dd)))))
    ))

;; This function creates the db tables
;;
(defun create-db-tables ()


  (db-create-table 'run
                   `((run_id i4 not-null)
                     (status (varchar ,*DB-RUN-STATUS-LENGTH*))
                     (step (varchar ,*DB-RUN-STEP-LENGTH*))
                     (step_path (varchar ,*DB-STEP-PATH-LENGTH*))
                     (log_id i4 not-null)
                     (lock (varchar ,*DB-RUN-LOCK-LENGTH*))
                     (owner_id i4 not-null)
                     (name (varchar ,*DB-RUN-NAME-LENGTH*))
                     (cf_id i4 not-null)
                     (rf_id i4 not-null)
                     (obj_id_gen i4 not-null)
                     (layer_seq i4 not-null)
                     (mask_set_id i4 not-null)
                     (mask_seq i4 not-null)
                     (pif_print_seq i4 not-null)
                     (pf_id i4 not-null)
                     (lot_size i4 not-null)))
  (db-modify 'run
             'cbtree
             :on '(run_id)
             :unique 'yes)

  (db-create-table 'evaluation_frame
                   `((run_id i4 not-null)
                     (frame_id i4 not-null)
```

```
                    (extend i4 not-null)
                    (frame_type (varchar ,*DB-FRAME-TYPE-LENGTH*)
                                with-null)
                    (frame_slots (varchar ,*DB-FRAME-SLOTS-LENGTH*)
                                 with-null))
                  :duplicates nil)

(db-modify 'evaluation_frame
           'cbtree
           :on '(run_id frame_id extend)
           :unique 'yes)

(db-create-table 'user_dialog
                 `((run_id i4 not-null)
                   (id i4 not-null)
                   (extend i4)
                   (name (varchar ,*DB-LOG-NAME-LENGTH*))
                   (arguments (varchar ,*DB-LOG-ARG-LENGTH*))
                   (step_path (varchar ,*DB-STEP-PATH-LENGTH*))
                   (tag (varchar ,*DB-LOG-TAG-LENGTH*))
                   (procedure (varchar ,*DB-LOG-PROC-LENGTH*))))

(db-modify 'user_dialog
           'cbtree
           :on '(run_id id extend)
           :unique 'yes)

(db-create-table 'wip_log
                 `((run_id i4 not-null)
                   (id i4 not-null)
                   (extend i4 not-null)
                   (user_id i4 not-null)
                   (name (varchar ,*DB-LOG-NAME-LENGTH*))
                   (time date)
                   (step_path (varchar ,*DB-STEP-PATH-LENGTH*))
                   (comment_p i4 not-null)
                   (tag (varchar 20))
                   (procedure (varchar 40))
                   (arguments (varchar ,*DB-LOG-ARG-LENGTH*))))

(db-modify 'wip_log
           'cbtree
           :on '(run_id id extend)
           :unique 'yes)

(db-create-table 'wip_log_comment
                 `((run_id  i4 not-null)
                   (id i4 not-null)
                   (line_num i4 not-null)
                   (comment (varchar ,*SQL-COMMENT-LENGTH*))))

(db-modify 'wip_log_comment
           'cbtree
           :on '(run_id id line_num)
           :unique 'yes)

(db-create-table 'wip_user              ; can't call the table user because
                                        ; of sql syntax conflict.
                 `((id i4 not-null)
```

```
                (name (varchar ,*DB-USER-NAME-LENGTH*))
                (address (varchar ,*DB-MAIL-ADDRESS-LENGTH*))
                (editor (varchar ,*DB-UDEFAULT-LENGTH*))
                (status (varchar ,*DB-UDEFAULT-LENGTH*))
                (tracing (varchar 3))
                (runidq (varchar ,*DB-UDEFAULT-LENGTH*))
                (nameq (varchar ,*DB-UDEFAULT-LENGTH*))
                (statusq (varchar ,*DB-UDEFAULT-LENGTH*))
                (proc_flowq (varchar ,*DB-UDEFAULT-LENGTH*))
                (stepq (varchar ,*DB-UDEFAULT-LENGTH*))
                (ownerq (varchar ,*DB-UDEFAULT-LENGTH*))))

(db-modify 'wip_user
           'cbtree
           :on '(id)
           :unique 'yes)

(db-create-table 'run_user
                 '((run_id i4 not-null)
                   (user_id i4 not-null)))

(db-modify 'run_user
           'cbtree
           :on '(run_id user_id)
           :unique 'yes)

(db-create-table 'mask_set
                 `((id i4 not-null)
                   (name (varchar ,*DB-MASK-SET-NAME-LENGTH*))))

(db-modify 'mask_set
           'cbtree
           :on 'id
           :unique 'yes)

(db-create-table 'mask
                 `((mask_set_id i4 not-null)
                   (id i4 not-null)
                   (number i4 not-null)
                   (extend i4 not-null)
                   (name (varchar ,*DB-MASK-NAME-LENGTH*) not-null)
                   (type (varchar ,*DB-MASK-TYPE-LENGTH*) not-null)
                   (location (varchar ,*DB-MASK-LOCATION-LENGTH*) not-null)))

(db-modify 'mask
           'cbtree
           :on '(mask_set_id id)
           :unique 'yes)

(db-create-table 'material
                 `((run_id i4 not-null)
                   (id i4 not-null)
                   (extend i4 not-null)
                   (primary (varchar ,*DB-MATERIAL-PRIMARY-LENGTH*))
                   (rest (varchar ,*DB-MATERIAL-REST-LENGTH*))))

(db-modify 'material
           'cbtree
           :on '(run_id id extend))
```

```
(db-create-table 'layer
                 `((run_id i4 not-null)
                   (id i4 not-null)
                   (name (varchar ,*DB-LAYER-NAME-LENGTH*) not-null)
                   (extend i4 not-null)
                   (definition (varchar ,*DB-LAYER-DEFN-LENGTH*) not-null)
                   (cache (varchar ,*DB-LAYER-CACHE-LENGTH*) not-null)))

(db-modify 'layer
           'cbtree
           :on '(run_id id)
           :unique 'yes)

(db-create-table 'lot
                 `((run_id i4)
                   (id i4)
                   (extend i4)
                   (name (varchar ,*DB-LOT-NAME-LENGTH*))
                   (bits i4)
                   (lsb i4)))

(db-modify 'lot
           'cbtree
           :on '(run_id id)
           :unique 'yes)

(db-create-table 'wafer
                 `((run_id i4)
                   (id i4)
                   (extend i4)
                   (ndex i4)          ; can't use index since it is SQL token
                   (snapshot_id i4)
                   (scribe (varchar ,*DB-WAFER-SCRIBE-LENGTH*))))

(db-modify 'wafer
           'cbtree
           :on '(id)
           :unique 'yes)

(db-create-table 'pif
                 `((run_id i4)
                   (print_name i4)
                   (type (varchar ,*DB-PIF-TYPE-LENGTH*))))
(db-modify 'pif
           'cbtree
           :on '(run_id print_name)
           :unique 'yes)

(db-create-table 'pif_snapshot
                 `((run_id i4)
                   (print_name i4)
                   (extend i4)
                   (parent i4)
                   (segments (varchar ,*DB-SNAPSHOT-SEGMENTS-LENGTH*))
                   (attrhash (varchar ,*DB-SNAPSHOT-HASH-LENGTH*))
                   (revhash (varchar ,*DB-SNAPSHOT-HASH-LENGTH*))
                   ))
```

```
(db-modify 'pif_snapshot
           'cbtree
           :on 'run_id)

(db-create-table 'pif_boundary
                 `((run_id i4)
                   (print_name i4)
                   (upper i4)
                   (lower i4)))

(db-modify 'pif_boundary
           'cbtree
           :on '(run_id))

(db-create-table 'pif_attr
                 `((run_id i4)
                   (print_name i4)
                   (extend i4)
                   (name (varchar ,*DB-PIF-ATTR-NAME-LENGTH*))
                   (value (varchar ,*DB-PIF-ATTR-VALUE-LENGTH*))))

(db-modify 'pif_attr
           'cbtree
           :on '(run_id))


(db-create-table 'process_flow
                 `((id i4 not-null)
                   (name (varchar ,*DB-PF-NAME-LENGTH*) not-null)
                   (current_version (varchar ,*DB-PF-VERSION-LENGTH*))))

(db-modify 'process_flow
           'cbtree
           :on '(id)
           :unique 'yes)

(db-create-table 'module
                 `((id i4 not-null)
                   (type (varchar ,*DB-PF-TYPE-LENGTH*) not-null)
                   (version (varchar ,*DB-PF-VERSION-LENGTH*))
                   (version_tag (varchar ,*DB-PF-VERSION-LENGTH*))
                   (name (varchar ,*DB-PF-NAME-LENGTH*) not-null)
                   (comment (varchar ,*DB-PF-COMMENT-LENGTH*))
                   (time date)
                   (status (varchar ,*DB-PF-STATE-LENGTH*))
                   (user_id i4)
                   (owner_id i4 not-null)
                   (use_counter i4 not-null)
                   (file (varchar ,*DB-PF-FILE-NAME-LENGTH*) not-null)))

(db-modify 'module
           'cbtree
           :on '(id)
           :unique 'yes)

(db-create-table 'pf_comment
                 `((pf_id i4 not-null)
                   (line_num i4 not-null)
```

```
                (comment (varchar ,*DB-PF-COMMENT-LENGTH*))))

(db-modify 'pf_comment
           'cbtree
           :on '(pf_id line_num)
           :unique 'yes)


(db-create-table 'pf_arg
                 `((pf_id i4 not-null)
                   (arg_num i4 not-null)
                   (name (varchar ,*DB-PF-ARG-NAME-LENGTH*))
                   (default_value (varchar ,*DB-PF-ARG-DEF-LENGTH*))))

(db-modify 'pf_arg
           'cbtree
           :on '(pf_id arg_num)
           :unique 'yes)


(db-create-table 'procedure
                 `((module_id i4 not-null)
                   (extend i4 not-null)
                   (name (varchar ,*DB-FUNCTION-NAME-LENGTH*)
                         with-null)
                   (code (varchar ,*DB-FUNCTION-CODE-LENGTH*)
                         with-null)))

(db-modify 'procedure
           'cbtree
           :on '(pf_id name extend)
           :unique 'yes)
)
```

[This page intentionally blank]

Appendix E
WIP Interpreter Data Structures

E.1 Introduction

This document contains Common Lisp definitions for some of the structures and classes used to represent run state. Complete definitions of all data structures may be found in the WIP interpreter source code.

E.2 Structure Definitions

```
(defstruct run
  current-frame                    ; The evaluation frame at the top
                                   ;   of the stack
  root-frame                       ; The evaluation frame at the stack
                                   ;   base
  id                               ; Integer identifying this run
  (object-id-generator 0)          ; Generator for object ids
  (current-log-id 0)               ; Generator for WIP-log ids
  (bindings (bpfl-bindings-init))  ; Global variable bindings
  (rework-lot-id-generator 0)      ; Rework lot id generator
  exception-frames                 ; List of frames at the base of
                                   ;  exception branches
  module-id-list                   ; Modules used by the run
  step-path                        ; Current step-path
  (wafer-lot-state
   (bsys::make-wafer-lot-state))   ; Representation of wafers and lots
  (materials
   (bsys::make-material-state))    ;
  (layers
   (bsys::make-layer-state))       ;
  (masks
   (bsys::make-mask-state))        ;
  snapshots                        ;
  )

(defclass ROOT-FRAME ()
    ((action                       ; the dispatch function
       :accessor frame-action
       :initarg :action
       :initform #'go-root)
     (next-action
       :accessor frame-next-action
       :initarg :next-action
       :initform #'go-root)
     (code                         ; lisp (list) code
       :accessor frame-code
       :initarg :code
       :initform nil)
     (cp                           ; position with code
```

```
      :accessor frame-cp
      :initarg :cp
      :initform (list 1))
     (returned-values                    ; list of values from child frame
      :accessor frame-returned-values
      :initarg :returned-values
      :initform nil)
     (children
      :accessor frame-children
      :initform nil)
     (id
      :accessor frame-id
      :initarg :id
      :initform (incf (run-object-id-generator *CURRENT-RUN*)))
     ))

(defclass EVAL-FRAME (root-frame)
    ((parent                             ; the frame that created this one
      :accessor frame-parent
      :initarg :parent
      :initform nil)
     (parent-cp
      :accessor frame-parent-cp
      :initarg :parent-cp)
     (code-package
      :accessor frame-code-package
      :initarg :code-package)
     (lex-bindings                       ; the lexical stack
      :accessor frame-lex-bindings
      :initarg :lex-bindings)
     (recipient                          ; the frame to receive eval result
      :accessor frame-recipient
      :initarg :recipient)
     (current-lot-name
      :accessor frame-current-lot-name
      :initarg :current-lot-name))
    )

(defstruct WAFER-LOT-STATE
  (lot-defns (let ((h (make-hash-table :test #'eq)))
               (setf (gethash 'bpfl::*ALL-WAFERS* h)
                 (make-lot :id 1))
               h))                       ; Lot-defns is hash table.  Key is
                                         ;  lot-name, value is lot-struct.
  (wafer-hash (make-hash-table))         ; Key is wafer-id, value is wafer
                                         ;   object
  (last-wafer 0)                         ; Wafer index generator
  (last-lot 1))                          ; Lot id generator

(defclass LOT ()
  ((id :reader lot-id
       :initarg :lot-id
       :initform (incf (*LAST-LOT*)))
  (bits 0))

(defclass WAFER ()
    ((snapshot :accessor wafer-snapshot
               :initarg :snapshot)
```

```
      (id :reader wafer-id
          :initarg :id)
      (index :reader wafer-index
             :initarg :index
             :initform (incf (*WAFER-INDEX*)))
      (scribe :reader wafer-scribe
            :initarg :scribe))


(defstruct MATERIAL-STATE
  (known-materials (make-hash-table :test #'eq)))

(defclass MATERIAL ()
  ((primary :reader material-primary    ; material name (symbol)
            :initarg :primary)
   (attrs :reader material-attrs-slot   ; material attributes
          :initarg :attrs)
   (list :reader material-list          ; generator for material
         :initarg :list))
  (:default-initargs :attrs nil :list '("?")))



(defclass LAYER ()
    ((CUBES :reader layer-cubes
            :initarg :cubes)
     (PRINT-FORM :accessor layer-print-form
                 :initform nil))
  )



(defstruct LAYER-STATE
  *KNOWN-LAYERS*                        ; plist layer-name -> original definition
  *LAYER-CACHE*                         ; plist layer-name -> precomputed cubes
  *LAYER-NUM-NAME*                      ; alist layer-number -> name
  (*LAYER-SEQ* 0)                       ; stacking order of layers
  )

;; default system layer definitions

(deflayer :TOP nil)
(defsysvar *TOP-SIDE* (layer :top))

(defbpfl TOP-SIDE ()
  *top-side*)

(defclass MASK ()
    ((NAME :reader mask-name
           :initarg :name)
     (ATTRS :reader mask-attrs
            :initarg :attrs)
     (SEQ :reader mask-seq
          :initform (incf (*mask-seq*))))
  (:default-initargs :location nil))

(defstruct MASK-STATE
  *KNOWN-MASKS*                         ; plist name -> mask object
```

```
  (*MASK-SEQ* 0)                        ; stacking order of masks
  )

;; A place-holder class.  Defines what can have PIF-ATTR's attatched to it.
;;
(defclass PIF ()
    ((PRINT-NAME :reader pif-print-name
                 :initarg :print-name))
  (:default-initargs :print-name (incf (*pif-print-seq*))))

;; Binding environment for attributes and boundaries.
;; All associations between objects are within the context of a snapshot.
;; A snapshot is attached to each wafer of interest to represent its
;; current state.
;;

(defclass SNAPSHOT (pif)
    ;; the most recent checkpoint for this snapshot.
    ((PARENT :reader snapshot-parent
             :initarg :parent)
     ;; a cache of component segments
     (SEGMENTS :accessor snapshot-segments
               :initarg :segments)
     ;; key is object, value is list of attatched attrs and boundaries.
     ;; objects are present in the snapshot if present in this table.
     (ATTR-HASH :reader snapshot-attr-hash
                :initform (make-hash-table :test #'eq))
     ;; key is attr, value is list of objects attatched to it in this ss.
     (REV-HASH :reader snapshot-rev-hash
               :initform (make-hash-table :test #'eq)))
  (:default-initargs
      :parent nil :segments nil))

(defclass SEGMENT (pif)
    ;; Segments exist to have things attached to them
    ()
  )

;; Boundaries may be used at most once in a snapshot.
;; Attributes may be attatched to distinct objects within a snapshot.
;;
(defclass PIF-ATTR (pif)
    ((NAME :reader pif-attr-name
           :initarg :name)
     (VALUE :reader pif-attr-value
            :initarg :value))
  (:default-initargs :value nil))

(defclass BOUNDARY (pif)
    ;; these 'attributes' on a boundary are required
    ((UPPER :reader boundary-upper
            :initarg :upper)
     (LOWER :reader boundary-lower
            :initarg :lower))
  )
```

```
handler-case
  rework-body;
  if (not rework-test) then
    signal-exception('force-rework);
  end;
  on-exception c := force-rework do
    /* invoke rework */
    decrement count;
    if count < 0 then
      retry-failure
    else
      push-rework-branch();
      with-lot 'rework begin
        rework-prefix;
        restart-body();
      end;
    end;
  end;
  on-exception c := merge-rework do
    /* merge rework branches */
    merge-rework-branches();
    ignore-exception();
  end;
end;
```

Figure 5-16: Rework implementation.

exception handler (see Table 5-12). `Ignore-exception` discards the exception and continues execution as if it had never occurred. `Restart-body` forces the code within the body of the **handler-case** to be executed again. `Resignal` forces the exception to be handled by the next higher-level exception handler in the code. In other words, the exception is passed to a higher-level section of code. `Halt-run` displays a user-dialog that allows the user to decide which of the above actions to take.

Rework is implemented using **handler-case** as shown in Figure 5-16. Two exception-handlers are defined in Figure 5-16: `force-rework` and `merge-rework`. The body of the **handler-case** executes the body of the **rework-loop** statement (i.e., *rework-body*), followed by the *rework-test*. If the test fails, it raises a `force-rework` exception. This exception can also be

Operation	Description
`ignore-exception`	Continue run execution.
`restart-body`	Restart the code in the body of the handler-case.
`resignal`	Pass the exception up to the next higher-level exception handler in the code.
`halt-run`	Save run, display user-dialog allowing the user to chose the desired action.

Table 5-12: Exception-handler operations.

raised within the *rework-body* code from the UI process or by a `raise-exception` procedure call.

When a `force-rework` exception is raised, the code in the `force-rework` handler is executed. The first action is to decrement the **`rework-count`** and halt the run if the count is exceeded. The current execution frame state is saved using `push-rework-stack`, and the current-lot is set to `rework`. Then the code in **`rework-prefix`** is executed before the code within the *rework-body* is executed again. When the WIP interpreter process detects that the execution points of two or more rework branches are the same, it signals a `merge-rework` exception which merges the rework branches, and execution is allowed to continue.

Constraints are also implemented by the exception mechanism. For example, the following constraint:

```
constrain
  constraint-body;

  when test1 do
    case-code1;
  end;
  when test2 do
    case-code2;
  end;
end;
```

is implemented by the code in Figure 5-17. The first action is to set up the constraints specified in the constraint tests (e.g., *test1* and *test2*). This code communicates with a subsidiary program, called a *constraint-server*, to enforce the constraints. When constraints are violated, the server sends a `constraint-violation` signal to the WIP interpreter process which causes the code in the `constraint-violation` clause to execute. Once the *constraint-body* is complete, the con-

```
handler-case
  setup-constraints(test1,test2);
  constraint-body;
  remove-constraints(test1,test2);
  on-exception c := constraint-violation
    if test1 then
      case-code1;
    end;
    if test2 then
      case-code2;
    end;
  end;
end;
```

Figure 5-17: `Constrain` implementation.

straints are removed.

As an example of a constraint server, consider the implementation of timing constraints. When the WIP interpreter process executes a **constrain** statement, it first sets up the constraints. For example, in the pattern procedure in Figure 3-14, the timing constraint test is:

```
when (max-time-between('spin-on-resist, 'expose-resist, {2 day})
    or max-time-between('expose-resist, 'develop-resist, {1 hour})) do
  halt-run("time-constraint-violation in pattern");
end;
```

The setup action for this constraint is that the spin-on-resist, expose-resist and develop-resist procedures are tagged to indicate that they are used in constraints. The WIP interpreter process then proceeds to execute the code in the body of the **constrain**. The WIP interpreter process examines the tag when spin-on-resist is called. The tag indicates that the procedure is used in a max-time-between timing constraint which causes the WIP interpreter process to send a message to the timing-constraint server indicating that a timer of 2 days duration should be started.

If the WIP interpreter process reaches the expose-resist procedure before the timer expires, the tag on expose-resist causes the WIP interpreter process to send a message to the server telling it to cancel the timer, thus removing the timing constraint.

On the other hand, if the timer expires before the WIP interpreter process reaches the expose-resist procedure, the server signals a timing-constraint exception and the code in the **do** clause is executed.

Constraints can also be expressed in terms of an absolute starting time. For example, the constraint in the spin-on-resist procedure in Figure 3-15 is:

```
when max-time-between(last-dehyd-time,'resist-coat, {30 min}) do
  last-dehyd-time := dehydrate-wafers();
  lot('rework) := lot('current);
  raise-exception('rework);
end;
```

which uses a variable as the first argument to max-time-between. In this case, the setup action for the constraint is to start a timer that is set to expire 30 minutes after the value in last-dehyd-time and tag the resist-coat procedure. As before, the constraint is removed if the resist-coat procedure is called before the timer expires.

The final timing constraint implemented in the prototype system is based on equipment access times. For example, the constraint

```
when (max-time-between(last-equip-time(equipment: 'spinner),
                       last-equip-time(equipment: 'stepper), (2 day))
      or max-time-between(last-equip-time(equipment: 'stepper),
                       last-equip-time('equipment: developer), (1 hour))) do
  halt-run("time constraint violation in pattern");
end;
```

is similar to the constraint in Figure 3-14, except that the endpoints of the permissible delay are written in terms of equipment access times. The setup action for constraints involving equipment access times tags the equipment named in the constraint. When certain equipment activity occurs (e.g., the equipment is deallocated), any active constraint on the equipment is used to start a timer if the equipment is specified as the start-time of the constraint, or to halt a timer if the equipment is specified as the end time of a constraint.

To illustrate how other types of constraints may be implemented, consider the constraint in the following code:

```
constrain
    when current-temperature() > (22 degC) do halt-run(); end;
end;
```

This constraint could be implemented if a temperature sensor existed that can be read by a program. The setup action for the constraint is to instruct the server to signal the WIP interpreter process if the temperature exceeds 22 °C. The server could periodically read the temperature sensor and raise an exception if it exceeded 22 °C.

5.9 Version Control

The run management system in the UI process handles version control for process flows and run modification. Although these operations are invoked through the UI process, they are implemented by the WIP interpreter process. This section describes the version control system. Run modification is discussed in the next section.

Process flows are stored in text files that are managed by the Revision Control System (RCS) [51]. RCS locks BPFL modules to prevent simultaneous modification of the same module by different people. In addition, it stores the code modification tree in an efficient way so that multiple versions can be maintained. The system supports two types of modules: *flows* and *libraries*. Flows contain a BPFL procedure definition with the same name as the flow. When a run is created,

id (integer)	name (string)	type (string)	version (string)	version_tag (string)	user_id (integer)	owner_id (integer)
2	cmos-16	flow	1.0	initial		5
3	cmos-16	flow	1.1	split	42	5
4	litho	library	1.0	initial	42	5

time (datetime)	comment (string)
2/1/91 09:15	baseline cmos initial definition
2/5/91 11:22	added lot-split operations
2/5/91 11:22	standard kodak 820 resist lithography

Table 5-13: Module table definition and examples.

a flow is specified (e.g., cmos-16) and the run is started by calling the BPFL procedure with the same name as the flow. A library contains procedures and definitions that are used in process flows. The name of a library is normally chosen to identify the purpose of the procedures in the library (e.g., the litho library contains procedures used in photolithography).

RCS manages module locking by forcing users to *check-out* a module before they are permitted to modify it. When the user has completed modification, the module must be *checked-in* before another user can check it out.

Summary information about modules and versions is stored in the module table shown in Table 5-13. The id field stores an integer that uniquely identifies each module. The name field contains the name of the module, and the type field indicates whether the module is a flow or a library. The version field contains the RCS revision number of the module. The version_tag field can be used to assign a mnemonic version string (e.g., version 1.0 may have a version tag of initial). For checked-out modules, the user_id field contains the id of the user who has the module checked out. The owner_id field contains the id of the user who is responsible for maintaining the module. Only the module owner can change module permissions. The time field contains the last modification time of the module and the comment string gives a brief description of the module.

Every module has a list of users who are authorized to modify it. Flows also have a list of users who are permitted to start a run using the flow.

When a module is checked out, it is still available for use in runs unless the owner requests otherwise. When a module is checked in, the new code in the module can either be written on top

of the old code, or a new module can be created. For example, if the user who has checked out `cmos-16` version `1.1` in Table 5-13 checks in the code, he or she can either overwrite the old `cmos-16` version `1.1` with the new code or create a new version. The version of the new code in this case will be `1.2`, assuming that version `1.1` is the latest revision of `cmos-16`. If version `1.2` already exists, the new module will be `cmos-16` version `1.1.1` in accordance with RCS conventions.

Libraries simplify management of BPFL software because they provide a mechanism for sharing commonly used procedures across different process flows. For example, the `litho` library is used by most BPFL process flows, and it would be wasteful and inconvenient to maintain a copy of the same code in each process flow. A further advantage is that changes to the code in a library need only be made in one location. All runs using that library will use the new code.

A module indicates that it uses another library with the `requires` declaration. For example, a library that uses the `equipment` library might contain the code:

```
requires(equipment, version: latest);
```

The `version` argument indicates what version of the library to use, as explained in chapter 4. When a run is created, `requires` declarations are used to load all required libraries into the database so that the run can use them.

When a run is started, the following action occurs for the flow and all libraries used by the flow:

1. The UI process copies the flow used by the run into a file, and converts it to Lisp using a parser written in Lex [58] and Yacc [59]. The conversion from block-structured code to Lisp is described in more detail earlier in this chapter (see section 5.4).
2. The Lisp code is then parsed again by the WIP interpreter process for further syntax checking and macro expansion [39] in Lisp, and written to a file. Functions in the macro-expanded file are saved in the database. All other top-level declarations (e.g., `defmaterial`) are evaluated and the results of the evaluation are added to the definitions for the run.

The UI process uses Make [60] to prevent unnecessary parsing or macro-expanding of modules that have already undergone this process. Likewise, the WIP interpreter process only saves module pro-