AN OBJECT-ORIENTED DATABASE DESIGN FOR INTEGRATED CIRCUIT FABRICATION

by

Lawrence A. Rowe and Christopher B. Williams

Memorandum No. UCB/ERL M87/43

20 May 1987

AN OBJECT-ORIENTED DATABASE DESIGN FOR INTEGRATED CIRCUIT FABRICATION

by

Lawrence A. Rowe and Christopher B. Williams

Memorandum No. UCB/ERL M87/43

20 May 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

An Object-Oriented Database Design for Integrated Circuit Fabrication†

Lawrence A Rowe
Chistopher B Williams

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

Abstract

A database design for two applications that manage a semiconductor fabrication laboratory are described: a facility management system and a "work-in-progress" system. The database requirements generated by these applications are discussed and a solution is proposed that uses a next-generation database management system and an object-oriented programming environment.

1. Introduction

The management and control of an integrated circuit manufacturing facility requires databases that can manage conventional business data (e.g., accounting and inventory data) and engineering/scientific data (e.g., facility management data and measurement data collected during processing). Conventional database management systems cannot be used for these applications because they do not provide support for engineering/scientific data (e.g., geometric data types and imprecise data with units). Next-generation database systems currently being developed have been designed to support business, engineering, and scientific data [3,6,7,14,15,21,25]. This paper describes a database design for two applications that will be implemented on one of these next-generation systems to manage the Microelectronics Laboratory (MICROLAB) at the University of California, Berkeley. The development of these applications is part of a larger project on integrated circuit computer integrated manufacturing (IC-CIM) whose goal is to automate IC manufacturing [12].

† This research was supported by grants from the Semiconductor Research Corp., Fairchild Semiconductor, Harris Semiconductor, Philips/Signetics, and Siemens Corp., with a matching grant from the State of California's MICRO program.

Several applications have been developed to manage the MICROLAB using a commercially available relational DBMS [30]. These applications include: 1) a program to manage information on people authorized to use equipment in the laboratory, 2) accounting programs to bill contracts and grants for laboratory usage, and 3) a simple inventory control system to track material consumed during processing. The relational database system and associated application development tools that were used to implement these applications (e.g., the forms-based query/update interface, the report writer, and the embedded query language preprocessor for C) have been very successful. However, other applications have been developed that, although they used the DBMS, have not realized significant benefits from storing the data in the database. These applications include: 1) a facility management system and 2) a system to generate equipment recipes automatically for a particular class of processing [8]. A major reason for the lack of success was the absence of support for engineering/scientific data.

This paper describes a database design for two applications: a facility management system and a work in progress (WIP) system. These applications were chosen for three reasons. First, they solve serious problems associated with managing the MICROLAB. Second, analogous problems exist in other manufacturing systems so the technology developed can be applied elsewhere if we are successful. And third, the data managed by the applications is representative of the kinds of engineering/scientific data found in other problem domains. The applications will provide a good test of a next-generation database management system.

The applications will be implemented in OBJFADS [18] and the data will be stored in POSTGRES [25]. OBJFADS is an object-oriented programming environment for the development of interactive, multimedia database applications. Applications are coded in a conventional object-oriented notation. However, part of the object hierarchy is stored in a POSTGRES database so that the data and procedures can be shared by other applications [17]. OBJFADS provides high-level tools (e.g., windows, menus, forms, dialog boxes, etc.) for building graphical applications that run on a high performance workstation with a bitmapped display and mouse. The system is implemented in Common Lisp [23] and uses a standard window system ("The X Window System" [9]) and object model ("The Common Lisp Object System" (CLOS) [4]).

POSTGRES is a next-generation database system that provides several features to support engineering/scientific applications. First, the system allows users to define their own data types and access methods. Second, it

supports active databases (e.g., database alerters that can be used to signal an application when a value changes in the database [5] and triggers that can be used to spawn other database updates when a value changes [2]). Third, POSTGRES provides support for historical data (e.g., data can be archived to an optical disk and a query can access current data, historical data, or current and historical data). Lastly, the system provides support for rules [27] and data of type procedure [24] that will eventually be used in CIM applications but are not required for the applications discussed here.

This paper describes the database design for a facility management and WIP system application. The remainder of the paper is organized as follows. Section 2 describes the applications. Section 3 describes an object-oriented design model that is used to represent the database design and the data types that will be added to POSTGRES to support these applications. Sections 4 and 5 describe the noteworthy features of the database design for the two applications. A complete specification of the design is available from the authors. Section 6 concludes the paper with a description of the current status and future plans for the applications.

2. The Facility Management and WIP Applications

This section describes the two applications, the unusual data requirements they generate for a DBMS, and the features of POSTGRES that can be used satisfy these requirements.

The facility management system maintains information on all data representing the facility including: 1) spatial layout (e.g., the rooms, corridors, and closets), 2) equipment locations (e.g., furnaces, sinks, steppers, etc.), and 3) utility networks (e.g., electrical wiring, water pipes, and gas lines). In addition, the application keeps track of the connections between equipment and utility networks.

The main program in the facility management system is a "what you see is what you get" (WYSIWYG) editor that allows laboratory personnel to examine and update the facility data. Examples of the tasks that laboratory personnel must perform are: 1) find specific equipment and utility networks (e.g., "highlight the Tylan furnace" or "highlight the DI2 water network"), 2) find connections between equipment (e.g., "highlight all equipment connected to electrical circuit EC-4"), and 3) find entities within a particular area (e.g., "highlight all equipment within 6 feet of the NI gas line").

The interface to this application must be graphical and it must be possible for people to express queries without having to write a program. Moreover, it must be easy for unsophisticated computer users to learn the

system. The obvious solution is to develop a menu- and form-based interface to a graphical editor that displays a 2-dimensional view of the laboratory.[1] Figure 1 shows a sample interface that uses a Macintosh-like interface [1]. The menus (i.e., **Application**, **Help**, and **Query**) include operations to query and edit the facility database and to enter an on-line help system that explains how to use the application.

The data requirements of this application require the database system to store geometric data. Because the database for a moderately sized facility can be quite large, the system must support access methods and query processing strategies to implement spatial queries efficiently. Recent research in the database community suggests that solutions exist, they just need to be applied to a real application [11,16].

The second application is a "work in progress" (WIP) system that tracks lots being processed in the fabrication facility.[2] This system is composed of several applications including a WIP interpreter, a scheduler, a process-flow editor, and a laboratory manager tool. The WIP interpreter actually runs the laboratory. It instructs the equipment operators to execute the processing steps to be performed. Or, if the equipment is connected to the system, the WIP interpreter sends commands directly to the equipment to execute the steps automatically. It interprets a *process-flow* specification that describes how to manufacture a semiconductor. The process-flow specification describes the sequence of physical processes that are to be performed on the wafers. The interpreter also collects and stores measurements collected during processing that are used to analyze and monitor the manufacturing process and the products being produced.

The scheduler controls equipment allocation. The development of scheduling algorithms to maximize productivity in the presence of dynamically changing products, processes, and equipment is a major research topic

---

[1] A prototype version of this program has already been developed. Laboratory personnel and industrial visitors have been enthusiastic about the potential usefulness of the system.

[2] A *lot* is the unit of processing in semiconductor fabrication. It typically contains 25 wafers that are 4 inch diameter disks. Each wafer is broken up into 200 to 400 integrated circuits (i.e., chips). The wafers in a lot are held in a plastic carrier that is moved from station to station to be processed by different pieces of equipment. Some equipment has been automated to the extent that processing descriptions can be loaded into computers connected to the equipment to control processing. We call these descriptions *equipment recipes*.

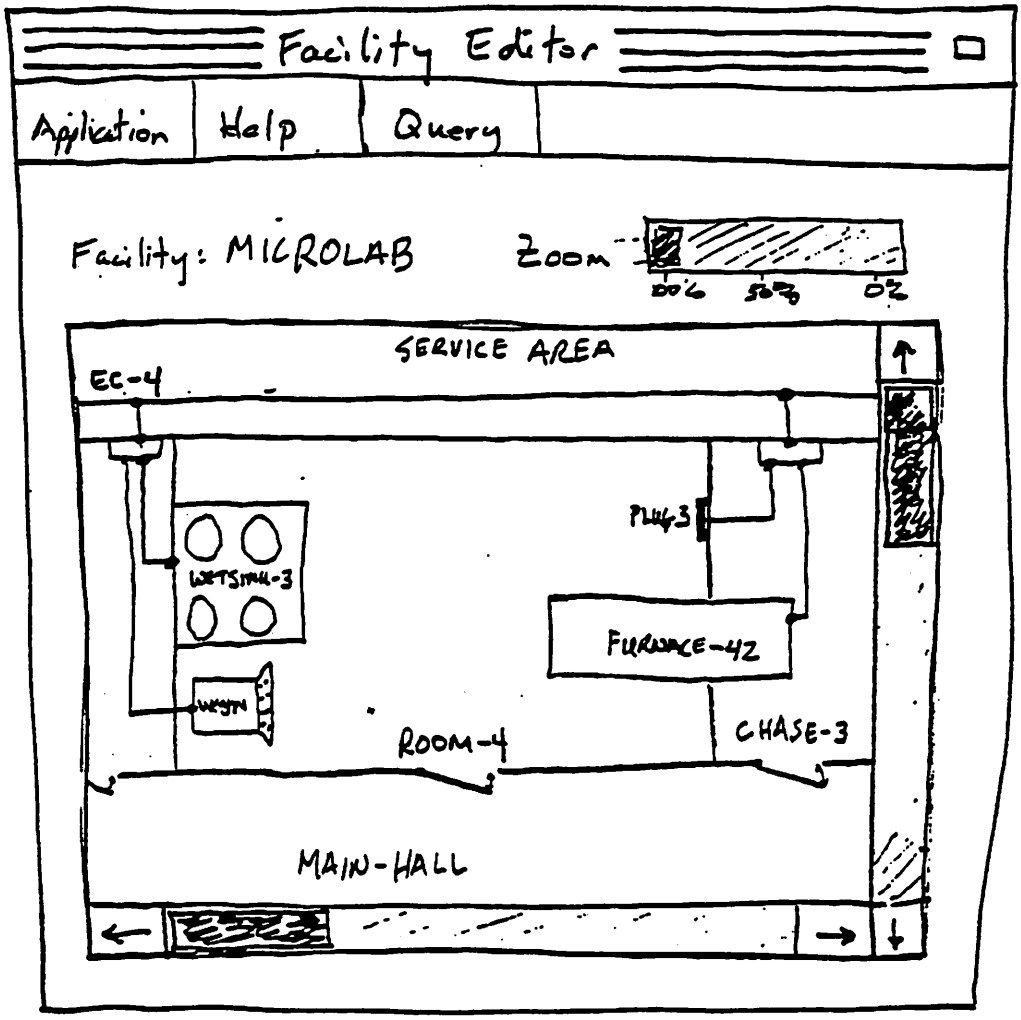


Figure 1: Facility editor interface.

in the semiconductor industry. This continuous, rapid change makes semiconductor manufacturing a particularly hard system to automate.

Process-flow specifications are coded in a language developed at Berkeley, called the *Berkeley Process-Flow Language* (BPFL), that is a mixture of declarative and procedural specification [20]. A program in this language is called a *process description* and a statement is called a *process step*. The declarative constructs describe the desired effects or goal of a processing step (e.g., "grow 1000 angstroms of oxide"). The procedural constructs

describe how to achieve the desired goal (e.g., "put the lot in tube 2 of the furnace and run recipe 22"). Procedures are divided into generic equipment procedures (e.g., a furnace operation) and specific equipment procedures (e.g., a furnace operation implemented on a Tylan furnace) so that a process can be run on different pieces of equipment by substituting a different equipment specific procedure. BPFL is embedded into Common Lisp to simplify the implementation of the WIP interpreter and other applications that must read and interpret process specifications (e.g., process simulators and schedule planners)

The process-flow editor is essentially a structured editor for BPFL that allows process designers and laboratory personnel to enter and examine process descriptions. Figure 2 shows a sample interface to the process-flow editor. The editor interface presents different views of a process to the user simultaneously. The example in the figure shows: 1) the declarative specification of the *CMOS-Nwell* process, called the *process-flow view*, in the tool window, 2) a procedural definition, called a *module definition view*, in a second window, and 3) a detailed specification of a selected goal primitive, called the *flow primitive view*, in a third window. The procedure shown in the module definition view (*CMOS-init-oxide*) is a generic equipment procedure that implements the *grow*-primitive shown in the process-flow view. The flow primitive view displays detailed information on the selected primitive. This form-based view is used to enter process-flow primitives. The editor also provides access to a library of predefined procedures that can be queried to define or modify a process design.

Lastly, the laboratory manager tool is used to monitor all activities in the laboratory including lots being processed, equipment status, and processing and maintenance schedules. It also allows laboratory personnel to respond to exceptional conditions (e.g., an equipment failure or an emergency) by turning equipment on or off, initiating safety procedures (e.g., evacuate the laboratory), rearranging processing steps, or scheduling maintenance tasks.

All data collected and managed by the WIP applications will be stored in the database including:

1. process-flow specifications,
2. measurement data,
3. scheduling data,
4. equipment logs, maintenance reports, and manuals,

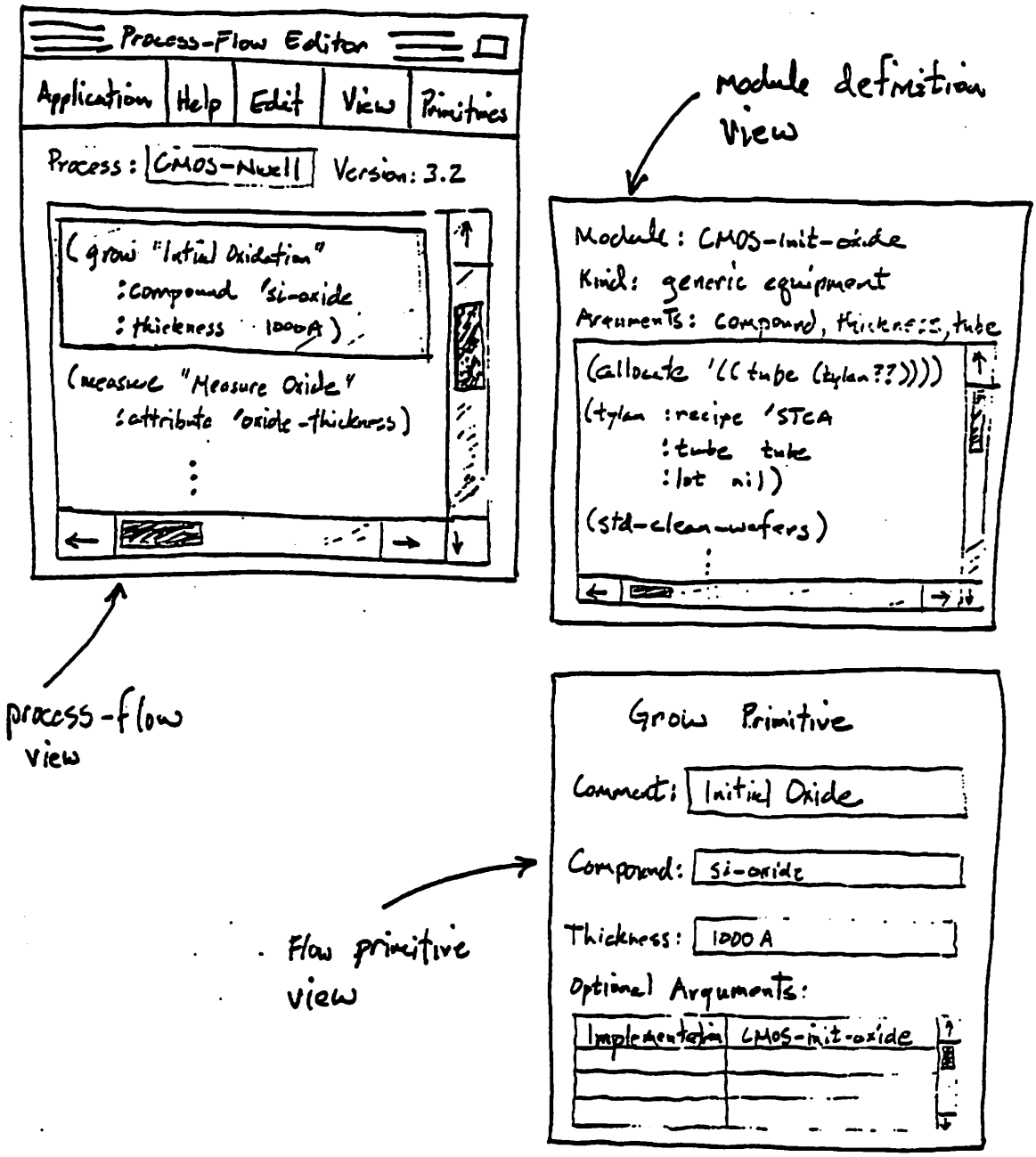


Figure 2: Process-flow editor interface.

5. the WIP log (e.g., the record of completed processing steps), and
6. WIP interpreter data (e.g., the location of all lots currently being processed, the wafers in a lot, etc.).

This data generates several unusual database requirements. First, process-flow specifications are essentially Common Lisp programs. As data, they can be viewed as complex objects (e.g., each procedure is an object) with shared subobjects (e.g., procedures are shared between different programs). However, the binding of objects to subobjects (i.e., the particular procedure called by a statement in another procedure) varies dynamically depending on the availability of equipment, the variations on the process being executed, and the process and equipment testing being conducted during manufacturing. Consequently, the database representation for process-flow specifications must allow these bindings to be changed dynamically without changing the text of the specifications.

The second unusual data requirement is the need to represent approximate values (e.g., measurements plus or minus an error term) and to keep track of the units involved (e.g., angstroms). Since computations are performed on this data, the database system must also support operators that automatically maintain error terms and that implicitly convert values to compatible units.

The last requirement is to maintain historical data. Many problems in semiconductor manufacturing concern changes over time (e.g., production yield, equipment availability, the processing history of a particular wafer, etc.) and the only way to solve them is to compare current measurements with past measurements. Consequently, an IC-CIM database must provide a convenient mechanism to manage historical data. Historical data may be gathered on a regular basis or it may be gathered to investigate a transient problem. In other words, it must be easy to turn on and off the collection of historical data without having to change the application programs that control the laboratory. The database system must also support easily expressed, but efficiently implemented, queries on historical data.

POSTGRES provides direct support to meet these requirements or mechanisms that allow application developers to extend the system to meet them. The POSTGRES data model is based on the relational model [19]. The relational model simplifies the design and maintenance of the database and it allows complex queries to be easily expressed. Current relational systems have acceptable performance for a wide range of applications and, as the commercial systems mature, they will compete favorably with any other

database system.

POSTGRES extends the relational model to allow developers to define their own data types, operators, and procedures so that application-specific data representations and algorithms can be supported in the database system. Moreover, the developer may define application-specific access methods (e.g., spatial access methods) so that queries on these data types can be efficiently implemented.

Finally, the POSTGRES storage system does not overwrite data [26]. An application designer can request the system to maintain a complete history of data in a relation, data since a particular date (e.g., "July 1, 1980"), or data in the recent past (e.g., "data in the relation in the last 30 days"). We believe that these mechanisms can be used to meet the application data requirements discussed above.

3. Design Model

This section describes the model that we have used for our database design and the data types that will be added to OBJFADS and POSTGRES to support the facility management and WIP applications.

The design model is similar to the model provided by an object-oriented programming language (OOPL) that supports multiple inheritance (e.g., the Common Lisp Object System [4]). We chose an object-oriented model for two reasons. First, the applications can be easily represented in an object-oriented language. The facility management and process-flow editors manipulate entities of varying types (e.g., rooms, equipment, procedures, etc.). Using object-oriented programming, the editors can be written by creating objects that represent the entities and associating the code with the object definitions. An object-oriented database design seemed natural because it closely resembles the data representation in the program. Another advantage of object-oriented programming is that programs are easier to modify which is important for IC-CIM applications because they are constantly being changed to accommodate new manufacturing technology and products.

The second reason we chose an object-oriented design model is that it simplifies the management of the design. The current database design has more than 50 object types and we expect that number to grow by a factor of 10 as more applications are added to the system. An object-oriented design will make this growth easier to manage.

The next few paragraphs describe the object-oriented design model. An *object* can be thought of as a record with named *slots*. Each slot has a data

type and a default value. The slot data type can be a primitive type (e.g., integer), a structured type (e.g., array), or a reference to another object.[3] The type of an object is called the object's *class*. Class information (e.g., slot definitions) is represented in another object called the *class object*.[4] A particular object is also called an *instance* and object slots are also called *instance variables*.

Procedures that manipulate objects, called *methods*, take arguments of a particular class (i.e., object type). Methods are associated with or bound to the class of their argument objects In most OOPL implementations method definitions are stored in the class object that represents the type of the argument object. Methods with the same name can be defined for different classes. For example, two methods named *area* can be defined: one that computes the area of a *box* object and one that computes the area of a *circle* object. The method executed when a program makes a call on *area* is determined by the argument class (i.e., type). For example,

area(x)

calls the *area* method for *box* if *x* is a *box* object or the *area* method for *circle* if it is a *circle* object. The selection of the method to execute is called *method determination*.

Classes inherit data definitions (i.e., slots) and method definitions (i.e., procedures) from other classes unless a slot or method with the same name is defined in the class. These other classes are called the *superclasses* of the class being defined and it is called a *subclass* of the other classes. Figure 3 shows a type hierarchy (i.e., class hierarchy) that defines equipment. The superclass of *Furnace* is the class *Equipment* so *Furnace* inherits all slots and methods defined for *Equipment*.

Figure 4 shows class definitions for *Equipment*, *Furnace*, and *Tylan*. An instance of a *Tylan* object inherits all slots defined for *Furnace* and *Equipment* (i.e., *Location*, *Picture*, *Date-Acquired*, *Number-of-Tubes*, and *Max-Temperature*). The only methods defined for *Tylan* are *Compile-Recipe* and *Load-Recipe* because *Furnace* and *Equipment* do not have any methods.

---

[3] Object references are represented by *object identifiers* (*objids*) that uniquely identify each object. Given an *objid*, the type of the object can be determined.

[4] The term *class* is used ambiguously to refer to an object type, the object that represents the type (i.e., the class object), and the set of instances of an object type. In this paper we will indicate the desired meaning in the surrounding text.

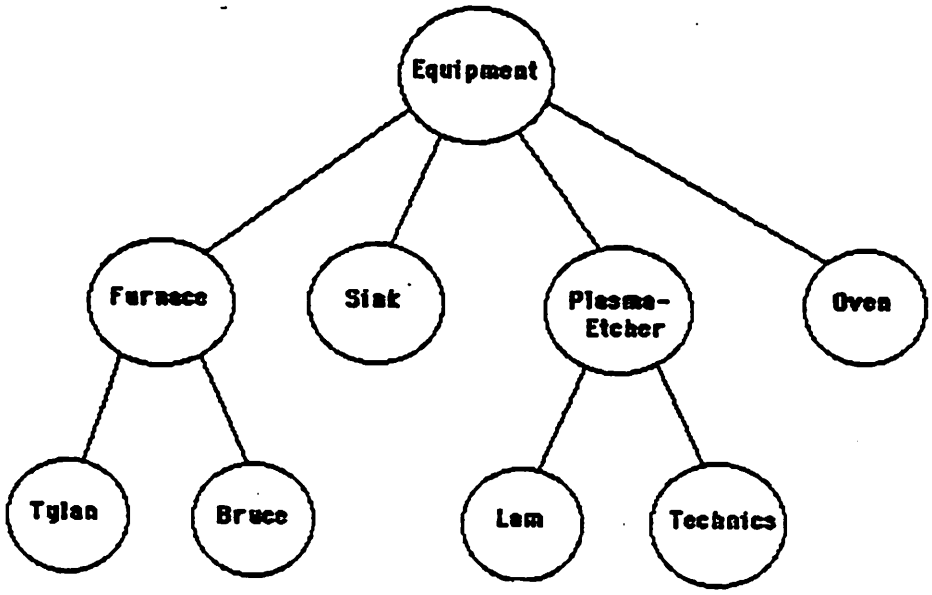


Figure 3: Equipment class hierarchy.

Class *Equipment*
Superclass *Object*
Instance Variables
Location
Picture
Date-Acquired
Methods

Class *Furnace*
Superclass *Equipment*
Instance Variables
Number-of-Tubes
Max-Temperature
Methods

Class *Tylan*
Superclass *Furnace*
Instance Variables
Methods
Compile-Recipe
Load-Recipe

Figure 4: Class definitions for equipment.

Slot and method definitions can be inherited from more than one superclass. For example, the *Tylan* class can inherit slots and methods that indicate how to communicate with the equipment through a network connection by including the *Network-Mixin* class in the list of superclasses.[5] Figure 5 shows the definition of *Network-Mixin* and the modified definition of *Tylan*. With this definition, *Tylan* inherits the slots named *Host-Name* and *Device* and the methods *Send-Message* and *Receive-Message* in addition to the slots inherited from *Furnace*. A name conflict arises if two superclasses define slots or methods with the same name (e.g., *Furnace* and *Network-Mixin* might both have a slot named *Status*). Name conflicts are resolved by inheriting the definition from the first class specified in the superclass list.

[5] The use of the suffix *Mixin* is a naming convention that indicates that this object defines behavior that is added to or mixed into other objects.

Class *Network-Mixin*
Superclass *Object*
Instance Variables
Host-Name
Device
Methods
Send-Message
Receive-Message

Class *Tylan*
Superclass *Furnace Network-Mixin*
Instance Variables
Methods
Compile-Recipe
Load-Recipe

Figure 5: Multiple inheritance example.

Inheriting definitions from multiple classes is called *multiple inheritance.*

The design model assumes the existence of conventional data types for slots including *integer*, *float*, *boolean*, *date/time* and arbitrary length *text* strings and *byte* arrays. The model also assumes the existence of geometric data types and type modifiers to support imprecise data with units.

Geometric data types are used to represent the physical and spatial properties of objects in a facility. Physical objects are specified in terms of a *base coordinate system.* The data type *point* defines a point in a three dimensional space. As described in the next section, objects are defined with these data types to represent graphical objects (e.g., lines and polygons) and application-specific objects (e.g., walls and rooms).

Most data maintained about semiconductor fabrication involves imprecise data. Data is imprecise because measurements are inexact or because a design objective is stated as an approximation of what is expected in practice. Consequently, a single number is not always appropriate to represent these values. Data types with accuracy errors (e.g., "3.05 ± 0.02") and interval values (e.g., "[3.03, 3.07]") are used to represent these cases. These data types are defined with the **inexact** and **interval** type modifiers. For example, the definition

Thickness: **inexact** integer

specifies that the *Thickness* slot has an inexact measurement of type *integer*. Operations are defined on these imprecise values that carry through the uncertainty or ignore it, depending on the needs of the application (e.g., assigning an inexact float value to an exact float slot discards the error term).

Data collected from the fabrication facility will often be subjected to statistical analysis. Consequently, null-value support is required. Null-values simplify the recording and analysis of missing or unknown data so that queries and aggregating functions can easily ignore inappropriate data. The type modifier **null** specifies that the slot may contain a null value.

Another characteristic of engineering/scientific data is that most data has an associated unit (e.g., microns, grams, meters per second, etc.) and dimension (e.g., length, volume, weight, temperature, etc.). Applications are easier to write if the programs can manipulate such data without having to know the specific units involved. For example, inventory from different storage areas can be added together to compute the total stock on hand without having to check for and perform unit conversions if the builtin operators do so automatically (e.g., "1 gallon + 1 quart = 1.25 gallon"). The type modifier **scaled** specifies that the slot value will have an associated unit specified and that the operators on the values should perform the required unit conversions.

An **array** type-constructor is provided to define structured data types. The elements of the array have the same type. They are accessed with an index number. For example, the definition

Vertices: **array**[point]

defines an array of *points*. Particular vertices can be referenced by

Vertices[i]

which returns the *i-th* vertex that is a value of type *point*.

4. Facilities Management Data Design

This section describes the database representation for the facility management data including: rooms, equipment, and utility networks.

A facility is represented by an instance of the class:

Facility(Name, Manager, Usage, Rating)

which has slots for the facility name, manager, the intended usage (e.g., *RESEARCH*, *PRODUCTION*, etc.), and rating (e.g., *CLASS-100*). All other

data about the facility is related to this object.

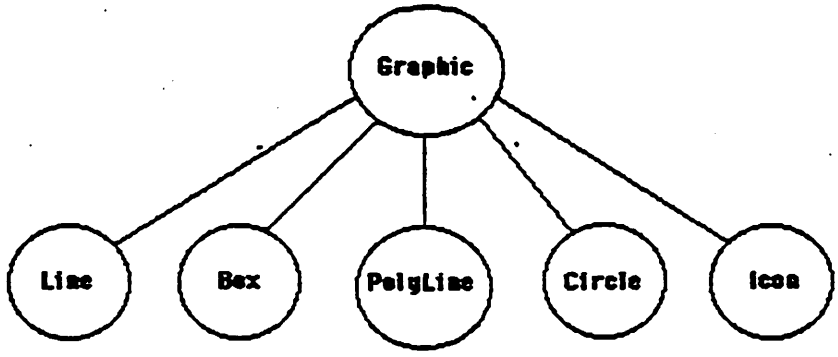
The facility layout uses a straightforward graphical representation. A hierarchy of graphical objects is defined that includes the primitives: *Line*, *PolyLine*, *Box*, and *Circle*. The hierarchy is shown in figure 6.

Walls and areas are used to represent rooms in the laboratory. They are defined by the following objects:

Wall(Facility-ID, Name, End-Points)
Area(Facility-ID, Name, Center, Clean-Level)

The slot *Facility-ID* is an object identifier for the facility that contains this wall or area.[6]

Walls and areas are given external names (e.g., "WALL-32" or "ROOM-4") to make it easier to relate the facility layout to the actual



Line(Start-Point, End-Point)
Box(Upper-Left, Width, Height)
PolyLine(Point-Array)
Circle(Center, Radius)
Icon(BitMap)

Figure 6: *Graphic* class hierarchy.

[6] We use the convention of appending the suffix *ID* to a class name to name a slot that contains a reference to an instance in that class. The value stored is an object identifier.

laboratory. The *End-Points* slot in *Wall* is an array of points that contain the two end points of the wall. The *Center* slot in *Area* is a point that specifies the center of the area. Each area in a laboratory must have a certain level of cleanliness. A value representing the required cleanliness is stored in the slot *Clean-Level.*

The *Wall* class has two subclasses that represent different kinds of walls: doors and windows. The *Area* class also has two subclasses that represent different kinds of areas: rooms and service chases. Typically, a service chase is located on the two opposing sides of a room to run utilities to equipment in the room without having to run it through the room. Slots are defined for these subclasses to represent the salient characteristics of each object. For example, *Door* has a slot that indicates whether the door has a window in it and *Room* has a slot to represent the safety level of the room.

Areas and walls must be related to complete the definition of an enclosed area. This relationship is represented by an object that maps areas to walls:

Area-Wall-Map(Area-ID, Wall-ID)

Notice that to describe a room, several instances of *Area-Wall-Map* are required (one for each wall).

Equipment in the laboratory is represented by pictures that are created from the graphical primitives. Pictures are represented by the objects:

Picture(Name, Represents)
Picture-Element(Picture-ID, Graphic-ID, Transform)

A *Picture* object describes a particular picture. It has two slots: the name of the picture (e.g., "Furnace") and a reference to the class object for the entity it depicts (e.g., the class object for a particular type of equipment).

The picture itself is represented by a collection of *Picture-Element* objects that specify the specific graphic object (e.g., lines, bitmaps, etc.). The *Transform* slot is an array that specifies a transformation of the element (e.g., translation, scaling, or rotation). *Picture* is also a subclass of *Graphic* so pictures can be built recursively. We plan to take advantage of a precomputation mechanism in POSTGRES to store a bitmap representation of the picture in a slot in each *Picture* object (not shown above) so that a picture can be fetched to display it without having to fetch all constituent elements [24].

A facility typically has many different utilities that must be represented in the database (e.g., electricity, deionized water, regular water, etc.). Each utility is represented by an instance of an object in the utility hierarchy shown in figure 7.

A utility network is essentially an undirected graph with points that represent equipment connections and junctions. A point in the graph with only one edge represents a utility source or sink or an equipment connection. A point with two or more edges is a junction (e.g., a circuit box in an electrical network). The network is represented by the following objects:

Junction(Utility-ID, Location, Picture-ID)
Connection(Source, Destination, Picture-ID)

An instance of *Junction* corresponds to a junction point in the graph. It has slots to represent the utility to which it belongs, its location (i.e., a *point* value), and a picture. An edge in the graph is represented by a *Connection* object that has slots to represent the source and destination and an optional picture for the connection (e.g., a pipe might be displayed as a sequence of shaded rectangles). The source and destination objects may be junctions or pieces of equipment.

Figure 8 show a small example of a utility network. The junctions (labelled "j*i*") are connected to a sink and a furnace. Figure 9 shows tables of instances that represent this example. The actual values in the tables

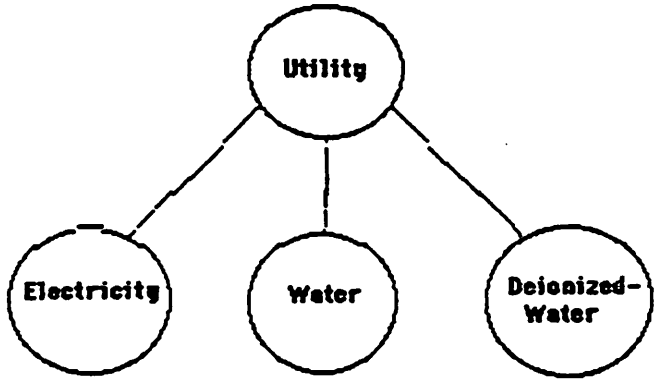


Figure 7: Utility network class hierarchy.

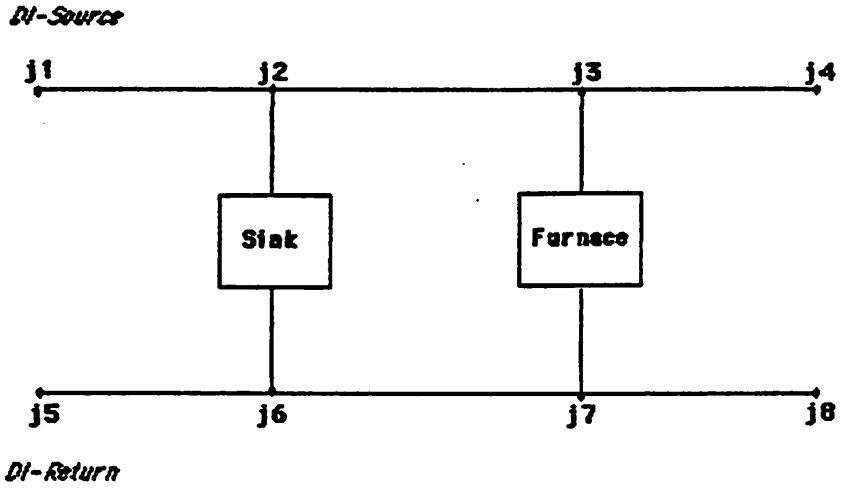


Figure 8: Utility network example.

Junction

Utility-ID	Location	Picture-ID
DI-Source	*j1*	...
DI-Source	*j2*	...
DI-Source	*j3*	...
DI-Source	*j4*	...
DI-Return	*j5*	...
DI-Return	*j6*	...
DI-Return	*j7*	...
DI-Return	*j8*	...

Connection

Source	Destination	Picture-ID
j1	*j2*	...
j2	*j3*	...
j3	*j4*	...
j5	*j6*	...
j6	*j7*	...
j7	*j8*	...
j2	*sink*	...
sink	*j6*	...
j3	*furnace*	...
furnace	*j7*	...

Figure 9: Utility example database representation.

are object identifiers. We show the name of the object as the value to make it easier to understand the example.

A database representation for the facility data will allow laboratory personnel to get the answers to many important question by querying the database. Some examples are:

1. What facilities have a plasma etcher?
2. What equipment is connected to electrical circuit "EC-4"?

3. What equipment is within 10 feet of the furnace in "ROOM-2"?

The query language must support transitive closure queries to solve the second question and the database system must support spatial access methods to solve the third question efficiently. The POSTGRES query language supports transitive closure queries and we plan to implement a spatial access method and add it to the system as a user-defined access method.

5. WIP System Data Design

This section describes the software architecture and database design for the WIP system applications including: process-flow specifications, state information for the WIP interpreter, and equipment status and scheduling data. At the end of the section, questions that can be easily answered by querying the database created by the WIP system are discussed.

The description of the WIP system is presented as though there was a single centralized database, and the WIP interpreter is presented as though there was a separate interpreter for each lot that is being processed (i.e., an interpreter per lot). While our initial implementation will have this architecture, we realize that many fabrication facilities require more concurrent users and higher WIP thruput than a centralized database and interpreter per lot architecture can achieve. Solutions to these problems already exist that we will use if and when we need better performance. The solution to the centralized database problem is to use a distributed database system that allows data to be distributed to different computers but still provides a centralized database view. In other words, applications can be written without having to know where the data is stored. Distributed database technology is well-understood and commercial products have recently become available [10,22,29].

The solution to the interpreter per lot problem is to have a single interpreter control many lots. This software architecture, called an *application server* architecture, is commonly used in high transaction rate on-line systems [13,28]. A "mini-operating system" is written to run in the application program that manages resources and multiplexes between several application tasks. In the WIP interpreter, an active lot (i.e., a lot currently being processed) corresponds to a task or process in a conventional operating system.

The application server architecture works well for applications that require transaction rates that can be achieved on a large, centralized computer. However, many factories will require much higher transaction rates.

The solution to this problem is to run several application servers at the same time (i.e., distributed application servers). In this architecture, the WIP interpreters can be located on a larger shared computer or distributed to computers at each work cell (i.e., operator workstation or equipment control computer). The two alternatives are shown in figure 10. In a centralized WIP interpreter, commands are sent to the cell computer to execute processing steps on a lot. Commands are sent to a different computer when the lot is moved to another cell. In a distributed WIP interpreter, the process-flow program is passed from one interpreter to another when the lot is moved [31]. We chose a centralized WIP interpreter architecture because the transaction rates in the MICROLAB are low and it simplifies the implementation. The database design changes required to implement a distributed WIP interpreter are relatively straightforward.

The remainder of the section describes the database design and questions that can be asked of the database.

5.1. Process-Flow Program Representation

Process-flow programs are essentially Common Lisp (CL) programs. CL programs are represented as lists of function calls which are themselves represented as lists.[7] For example, the function call "f(x,y)" is represented in CL as

```
(f x y)
```

A function call is also called a *symbolic expression* or *s-expression* (*s-exp*).

The class hierarchy for BPFL procedures is shown in figure 11. Slots are defined to represent the procedure *Name*, *Version*, and *Owner* (i.e., the person who created the procedure). The *Body* slot is a text data type that contains the procedure body. The body is represented by a Lisp lambda expression which specifies the formal arguments to the procedure and the sequence of s-expressions that constitute the code of the procedure. Subclasses of *BPFL-Proc* are defined for each procedure type: flow, generic equipment, and specific equipment. Notice that slots are only defined on *BPFL-Proc* and that they are inherited by the subclasses.

---

[7] A *list* is the primary data structure that Lisp programs manipulate. Consequently, program representations are the same as data representations. This ability to treat programs as data is one benefit of using Lisp as the host language for BPFL.

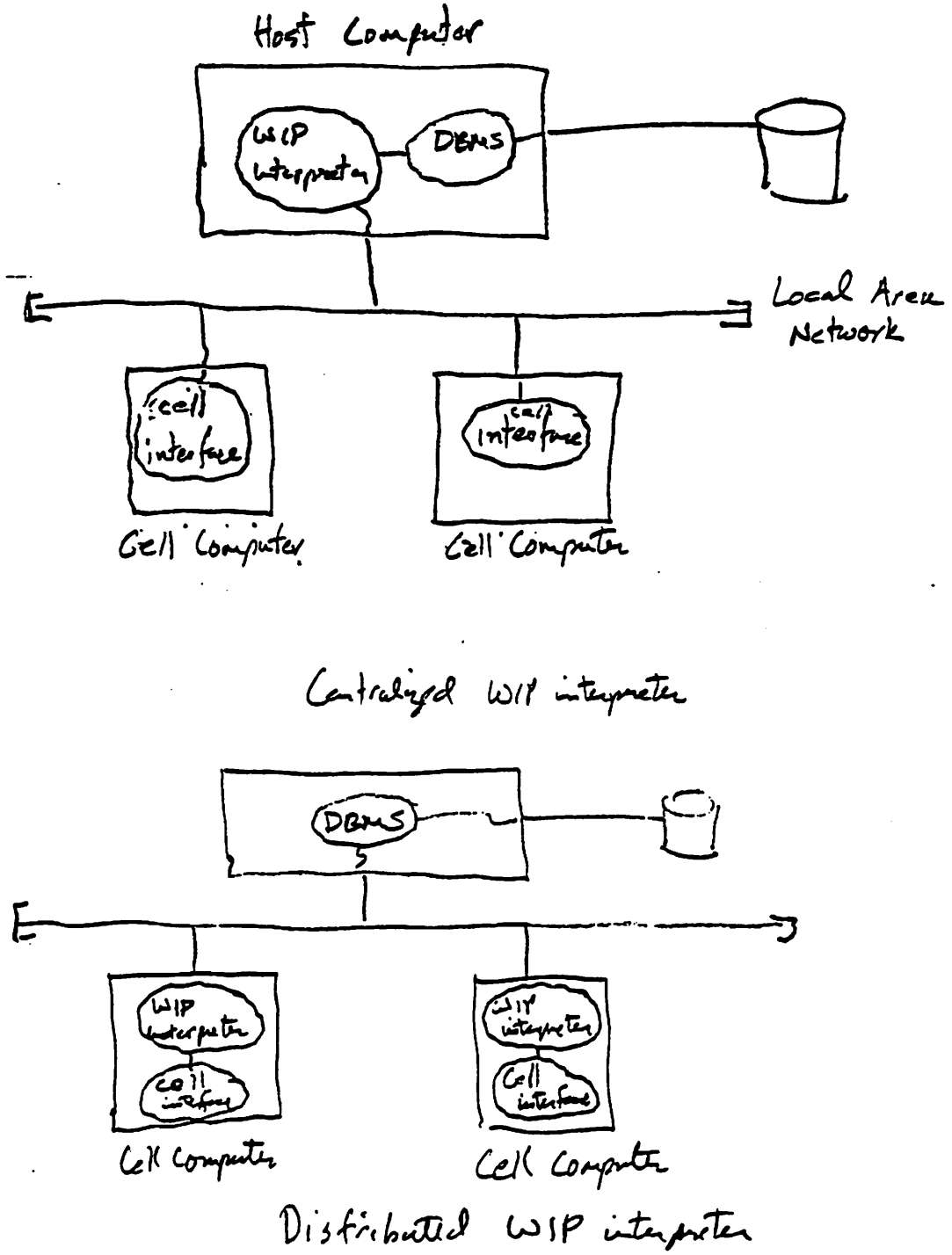
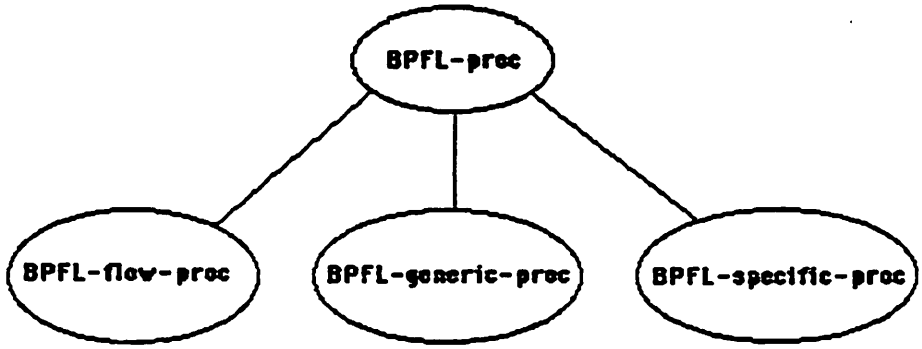


Figure 10: Centralized versus distributed WIP interpreter.



BPFL-Proc(Name, Version, Owner, Body)

Figure 11: BPFL procedure class hierarchy.

The WIP interpreter keeps track of the currently executing statement (i.e., s-expression) in a *BPFL-Proc*. In addition, it must have access to a symbol table of local variables declared in each procedure. The current s-expression is encoded as a list of integers that identify a particular function call in the procedure. For example, given the list

((s1) (s2) ((s3) (s4)) ...)

a reference to the s-expression containing *s4* is represented by

(3 2)

because it is the *2-nd* element in the *3-rd* element of the list.

The symbol table is represented by a class that maps a procedure identifier and variable name to a unique identifier. The class definition is

Symbol-Table-Entry (BPFL-Proc-ID, Variable-Name, Variable-Type)

The interpreter uses the *objid* for an instance of a *Symbol-Table-Entry* to access the current value of the variable in the data structures maintained by the WIP interpreter.

The term *process* has different meanings to the different people involved in IC manufacturing. To the production manager, it means a complete process to manufacture chips. To a process designer, it means a complete process or a step in a process. Process design engineers typically develop steps for a particular piece of equipment (e.g., LPCVD furnace operations) or complete process designs. Engineers want access to a library

of processes (complete processes and process steps) that have already been developed and tested. Consequently, the database representation for process-flow specifications must provide a process model that allows all users to view the data in a way that is compatible with the way they think about it.

Another problem with the design of a database representation for process-flow specifications is that it must be easy to change the equipment on which a particular process is run. For example, it should be possible to move a process to a different laboratory with different equipment without requiring a complete re-design of the process. By substituting the equipment specific procedures for the equipment in the new laboratory, the process should run. This type of change must also be possible while a lot is being processed because a particular piece of equipment might have failed or been replaced and you want to continue processing the lot.

We believe the solution to both problems is a very flexible mechanism to control which procedure is called by each process step that can be changed without having to modify the source code. In other words, the implementation of each process step can be arbitrarily changed. For example, figure 12 shows the definition of a process including the procedures that implement each step. The *defflow*, *defgeneric*, and *defgeneric* functions define flow, generic, and specific procedures, respectively. The comment (i.e., the text following the semicolon) indicates the procedure that implements each process step. For example, the *grow* step in *CMOS-Nwell* is implemented by *CMOS-init-oxide*. We call this mechanism *implementation binding* because it specifies the implementation of each procedure call in a process. All procedures, except WIP interpreter primitives, require an implementation binding.

Processes and implementation bindings are represented by the following two classes:

Process(Process-Name, BPFL-Proc-ID, IBind-Number)
IBind(BPFL-Proc-ID, S-Exp, Process-ID, IBind-Number)

Each process, whether it is a complete process or a process step, has a *Process* object that identifies the name of the process (*Process-Name*), the procedure that specifies the processing to be performed (*BPFL-Proc-ID*), and a set of implementation bindings for the procedure (*IBind-Number*). The implementation bindings are represented by *IBind* objects. Each *IBind* object indicates that a particular procedure call, specified by the slots *BPFL-Proc-ID* and *S-Exp*, is implemented by the process specified by the

```
(defflow CMOS-Nwell
   ...
   (grow "initial oxide" ...)          ; implemented by CMOS-init-oxide
   (measure "measure oxide" ...)       ; implemented by measure-oxide
   ...)

(defgeneric CMOS-init-oxide
   ...
   (furnace ...)                       ; implemented by tylan-run-sequence
   ...)

(defgeneric measure-oxide
   ...
   (visual-check-oxide ...)            ; implemented by operator-dialog
   (nano-spec-measure-oxide ...)       ; implemented by operator-dialog
   ...)

(defspecific tylan-run-sequence ...)

(defspecific operator-dialog ...)
```

Figure 12: Process definition with step implementations.

slot *Process-ID*. Notice that a process is composed of a procedure and a set of implementation bindings for the procedure.

Figure 13 shows how the example in the previous figure is represented. The column *Process-ID* is added to the *Process* table to show what entry is referenced by *Process-ID* in *IBind*. The tables show the binding for a process named *CMOS-1*. The implementations for the procedures called by *measure-oxide* are represented by instances of *IBind* with *IBind-Number* equal to 3. The process that implements the *visual-check-oxide* step is *Process* 5 (i.e., an *operator-dialog* call). To change this process to run on a different furnace, the *Process-ID* entry for *CMOS-init-oxide* in *IBind* is changed to a process that uses the other furnace. Of course, you may want to keep both processes around which requires that the *CMOS-1* entries in *Process* and *IBind* be copied and given a new name.

Although the representation of bindings complicates the database representation, we believe it will simplify the manipulation of process descriptions.

Process

Process-ID	Process-Name	BPFL-Proc-ID	IBind-Number
1	CMOS-1	*CMOS-Nwell*	1
2	CMOS-1	*CMOS-init-oxide*	2
3	CMOS-1	*tylan-run-sequence*	0
4	CMOS-1	*measure-oxide*	3
5	CMOS-1	*operator-dialog*	0

IBind

BPFL-Proc-ID	S-Exp	Process-ID	IBind-Number
CMOS-Nwell	*(grow ...)*	*2*	1
CMOS-Nwell	*(measure ...)*	*4*	1
CMOS-init-oxide	*(furnace ...)*	*3*	2
measure-oxide	*(visual-check-oxide ...)*	*5*	3
measure-oxide	*(nano-spec-measure-oxide ...)*	*5*	3

Figure 13: Implementation binding example.

5.2. WIP Interpreter State Representation

The WIP interpreter maintains database objects that represent runs, lots and wafers, the execution state of the currently active runs, and a log of steps executed and measurements collected during processing.

A class *Wafer* is defined to track wafers processed in the laboratory:

Wafer(Number, Orientation, Purity, O2, IsTest)

The *Number* slot is a unique number etched on the wafer so that it can be identified. The *Orientation*, *Purity*, and *O2* slots maintain information about the wafer. The *IsTest* slot indicates whether this wafer will be used to produce chips or to test the process and equipment. For example, in many processing steps a blank wafer will be run along with the production wafers to check the effect of the step. By using a blank wafer, the test will be independent of the effects of previous steps.

A lot may contain production wafers, production and test wafers, or just test wafers. A *logical lot* is a set of wafers that a BPFL procedure processes.

For example, the production wafers may be in one logical lot (e.g., the lot named *PRODUCT*) and test wafers may be another logical lot (e.g., a lot named *WELL-Control*). An optional argument to all BPFL procedures identifies by name the lot to which the procedure should be applied. The WIP interpreter must keep track of these different logical lots and the wafers they contain.

The WIP system interprets a process (i.e., a procedure and implementation binding) to manufacture or test wafers. Each distinct run that the WIP system executes can create many logical lots. These logical lots are held in *physical lots* that represent the plastic carriers that hold the wafers. The system must keep track of the logical lots, physical lots, and the wafers they contain. In addition, the system must keep track of the past history of these entities and relationships because lots can be split apart and merged back together and several logical lots can be mapped to one physical lot. Lot splitting and merging is used to control testing and to allow more than one product to be produced in the same run.

The database representation for this data is represented by the following classes that define runs, logical lots, and physical lots:

Run(Run-Number, Owner, Created-At, Process-ID)
Logical-Lot(Logical-Lot-Name, Run-ID)
Physical-Lot(Physical-Lot-Number, Status)

An instance of a *Run* object is created for each run in the laboratory. It identifies the *Run-Number*, the person who is responsible for the run (*Owner*), the date and time it was created (*Created-At*), and the process that was run (*Process-ID*). A *Logical-Lot* object is created for each logical lot created by a run. The two slots in this object identify the name of the lot (*Logical-Lot-Name*) and the run that created it (*Run-ID*). By convention, the logical lot named *PRODUCT* is the lot that contains the production wafers. The class *Physical-Lot* tracks each carrier for wafers. It identifies the serial number (*Physical-Lot-Number*) and the current status (e.g., *IN-USE* or *FREE*) of the carrier.

The mapping of logical lots to physical lots and wafers to positions in physical lots is represented by the following two classes:

Logical-Physical-Lot-Map(Logical-Lot-ID, Physical-Lot-ID, Start-Time, End-Time)
Wafer-Physical-Lot-Map(Wafer-Number, Logical-Lot-ID, Position)

The slots *Logical-Lot-ID* and *Physical-Lot-ID* in *Logical-Physical-Lot-Map* represent the mapping of logical lots to physical lots. The *Start-Time* and *End-Time* slots indicate when the logical lot was mapped into the physical

lot. The class *Wafer-Physical-Lot-Map* identifies the position of a wafer in a physical lot. It has slots to store the wafer number (*Wafer-Number*) and the position (*Position*) of the wafer in the carrier. The position is needed because some statistic analysis of process measurement data depends on whether the wafer was on the outside or inside of the carrier.

An example will make this design easier to understand. Suppose that a run was composed of three logical lots (*PRODUCT*, *TEST-1* and *TEST-2*), that *PRODUCT* had two wafers (*WAFER-1* and *WAFER-2*) and the two test lots each had a single wafer (*WAFER-3* and *WAFER-4*), and that *PRODUCT* and *TEST-1* were mapped to physical lot *PL-1* and *TEST-2* was mapped to physical lot *PL-2*. This data would be represented in the database by objects shown in the tables in figure 14. Entries in columns that hold object identifiers (i.e., slot names that end in *ID*) are shown as values in italics.

The state of lots currently being processed is maintained in three classes that correspond to the "process table" and "run-time stack" in a conventional operating system and programming language environment. The currently active runs are represented by instances of the class

Active-Runs(Run-Number, Stack-Frame-ID, State)

The *Run-Number* indicates the run, the *Stack-Frame-ID* indicates the current stack frame, and the current state of the run is stored in *State*. The state (e.g., *PROCESSING*, *WAITING*, or *SUSPENDED*) indicates that the lot is being processed by a piece of equipment, waiting for equipment, or suspended from processing (e.g., the student who is responsible for the run may be attending class).

The run-time stack for BPFL procedures is represented by instances of the following two classes:

Stack-Frame(BPFL-Proc-ID, Current-S-Exp, Parent-Stack-Frame-ID, LogIt)
Variables(Stack-Frame-ID, Variable-ID, Value)

A *Stack-Frame* object is created for each active BPFL procedure. The *BPFL-Proc-ID* and *Current-S-Exp* slots indicate the next process step to execute when this run is resumed. Local variables for the procedure are stored as instances of the class *Variables*. The *Parent-Stack-Frame-ID* identifies the stack frame of the procedure that called this one (i.e., the dynamic link in a conventional programming language stack frame). Lastly, the *LogIt* slot indicates whether the execution of the current procedure should be logged into the *WIP-Log*. If this slot has the value true, a WIP log record is

Run

Run-Number	Owner	Created-At	Process-ID
Run-1	Chris	15-April-87	*CMOS-Nwell*

Logical-Lot

Logical-Lot-Name	Run-ID
PRODUCT	*Run-1*
TEST-1	*Run-1*
TEST-2	*Run-1*

Physical-Lot

Physical-Lot-Number	Status
PL-1	IN-USE
PL-2	IN-USE

Logical-Physical-Lot-Map

Logical-Lot-ID	Physical-Lot-ID	Start-Time	End-Time
PRODUCT	*PL-1*	1530 15-April-87	
TEST-1	*PL-1*	1645 15-April-87	
TEST-2	*PL-2*	0840 16-April-87	

Wafer-Physical-Lot-Map

Wafer-Number	Logical-Lot-ID	Position
WAFER-1	*PRODUCT*	1
WAFER-2	*PRODUCT*	2
WAFER-3	*PRODUCT*	3
WAFER-4	*TEST-1*	4
WAFER-5	*TEST-2*	1

Figure 14: Database representation for lot mapping.

written that describes the execution of the procedure (e.g., the start and end times and the return result). The *LogIt* slot can be set by the person running the WIP interpreter or by a procedure call in the process. This

mechanism allows the granularity of logging to be changed dynamically.

The WIP interpreter receives events from a piece of equipment or an operator which causes it to perform some action on a run. The action can be to resume interpretation of a BPFL procedure, suspend execution of a run, or change the state of a run to *processing*. The event contains the *Run-Number* which the interpreter uses to retrieve the current state of the process (e.g., the stack frame). The interpreter then executes process steps in the procedure until an interpreter primitive suspends the run or enters a wait state pending completion of a processing step.

A transaction log is written by the WIP interpreter that provides a trace of processing steps initiated and completed. A hierarchy of log objects is defined that allows arbitrary log entries to be written. A sample hierarchy is shown in figure 15. Notice that all log records inherit the slots from *WIP-Log* which give a generic description of the logged event and that each subclass defines additional slots to represent information specific to that event represented by the subclass. The capability to dynamically add classes (e.g., new *WIP-Log* subclasses) makes the WIP system easier to develop and maintain.

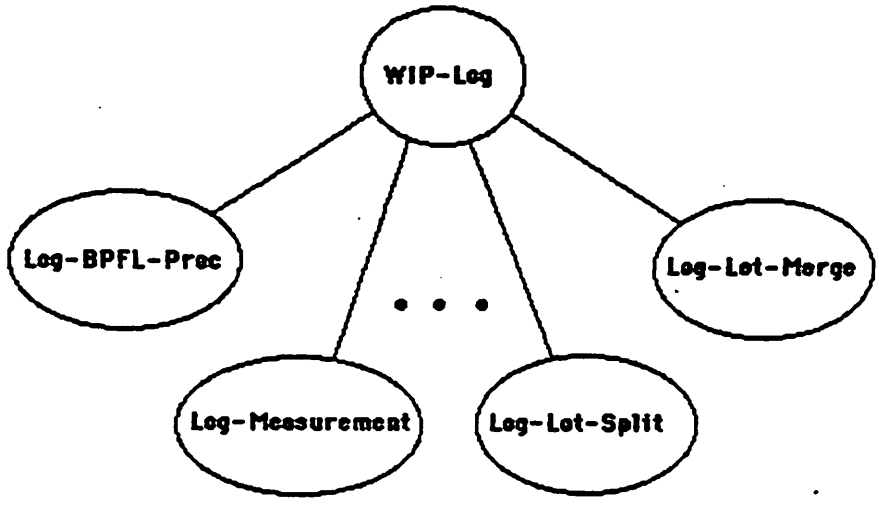
5.3. Equipment Scheduling Data Representation

This section describes the database representation for keeping track of equipment status and schedules.

The *Equipment* class contains information on all aspects of equipment including its current status, location in the facility, and display information for the facility editor. The class definition is:

Equipment(Status, Allocated-To, Estimated-Time-Available, Time-Last-Used, Facility, Location, Picture-ID, Orientation)

The *Status* slot indicates the current status of the equipment. Equipment can be available for use (*FREE*) or currently being used (*PROCESSING*). Equipment that is unavailable (e.g., some equipment can only be operating during normal working hours) has status *OFF-LINE*. And, equipment that is waiting to be repaired, currently being repaired, or being serviced as part of a regular maintenance program has status *WAITING-REPAIR*, *BEING-REPAIRED*, or *REGULAR-MAINTENANCE*. The *Allocated-To* slot indicates who has control of the equipment when it has been allocated. It contains the reservation identifier (described below) that indicates who reserved it.



WIP-Log(Run-Number, Physical-Lot-Number, Logical-Lot-Name,
Start-Time, End-Time)
Log-BPFL-Proc(Caller, S-Exp, Called-Proc, Args, Result)
Log-Measurement(Wafer-Position, Device, Measurement, Value)
...
Log-Lot-Split(Split-Lot-Name, Wafers-Split-Array)
Log-Lot-Merge(Logical-Lots-Merged, Result-Lot-Name)

Figure 15: *WIP-Log* hierarchy.

The *Estimated-Time-Available* slot indicates when the equipment will be free if is not currently available for processing (e.g., if it is off-line or being repaired). The *Time-Last-Used* slot is kept so that laboratory personnel can easily determine the recent use of a piece of equipment. This value is displayed in an equipment status summary that is displayed by the laboratory manager tool.

The *Facility*, *Location*, *Picture*, and *Orientation* slots store data about the location of the equipment and the graphical picture and orientation that the facility editor uses when displaying it. The *Picture* and *Orientation* slots use the data representations described in the previous section.

The scheduling programs also need information on equipment reservations. This data is kept in instances of the following class:

Equipment-Reservation(Equipment-ID, Reserved-At, Duration,

Usage, Reserved-By, Commenced-At)

The *Equipment-ID* slot identifies the piece of equipment covered by this reservation. The *Reserved-At* and *Duration* slots specify the date and time at which the equipment is to be reserved and the length of the reservation. The *Usage* slot indicates how the equipment will be used. Possible values for this slot are: *PROCESSING, MAINTENANCE, TRAINING,* and *OFF-LINE*. The *Reserved-By* slot indicates who made the reservation and the *Commenced-At* slot indicates when the reservation was used (e.g., when processing or maintenance began).

Reservations in the Berkeley MICROLAB are made by laboratory users with a simple reservation scheduling tool. In essence, the system works as a simple sign-up sheet. In a larger facility, elaborate scheduling programs generate reservations to maximize laboratory thruput. An additional class is needed that environment to keep track of the run that will be processed during the reservation period. A reservation may involve processing several runs at the same time in a piece of equipment. For example, several physical lots can be processed in a furnace at the same time. In a production facility, as many lots as possible are run at the same time because it significantly improves thruput.[8] The *Run-Reservation* class keeps track of this data:

Run-Reservation(Equipment-Reservation-ID, Run-ID)

This object maps a run (*Run-ID*) to a reservation (*Equipment-Reservation-ID*). Equipment scheduling algorithms will need other information (e.g., expected running times for the processing steps in a particular process). We expect this part of the database design to change considerably when these algorithms are added to the system.

The equipment status and reservation data is queried and updated by the WIP interpreter and a scheduling demon. When a process requests a piece of equipment, the WIP interpreter sends a message to the scheduling demon to request that it be allocated to the run. The scheduling demon determines if a reservation exists for the run to use the equipment. If not, a reservation can be defined if the equipment will be available during the required time. Assuming that the reservation exists, the scheduler demon sets the *Commenced-At* time for the reservation and notifies the WIP

[8] Furnace runs take many hours to complete.

interpreter that the equipment has been allocated to the run. The WIP interpreter receives the notification and resumes the execution of the process.

5.4. WIP Analysis

One advantage of storing the WIP system data in a database is that a variety of questions can be answered about the status of a laboratory and the past history of processing wafers by querying the database. This section describes example of the kinds of questions that can be answered. We do not present the actual queries because the important point is not how to express them but that they can be expressed.

Questions on three aspects of the WIP system database will be illustrated. First, questions can be asked about the past history of processing:

1. What other wafers were processed by the same furnace that did oxide deposition on *WAFER-123*?
2. What is the average yield for wafers processed by the *CMOS-Nwell* process?
3. Retrieve a complete processing history of *WAFER-123*.

The last question would generate a complete history of the processing steps, the results of measurements, and the logical and physical lots that contained the wafer during processing.

The second aspect of WIP data that can be queried is the current and past status of the laboratory. The following types of questions can be asked:

1. What percent of the time has the plasma etcher been free during normal office hours in the past 10 working days?
2. What was the weekly availability (i.e., percentage of time free) of all furnaces in the last 6 months.
3. Which equipment has been unavailable the most time because of maintenance or repair this week?

The last aspect of the WIP system data concerns future activities and plans. OBJFADS and POSTGRES allow versions of object classes to be created. These versions can be updated and queried. A run simulator could be written that simulates laboratory processing under various constraints (e.g., improved equipment reliability, addition of new equipment, or changes in processing times). After the simulator has been run, the hypothetical log (i.e., the version of the WIP log created by the simulator) can be queried to determine the improvements in yield and equipment utilization. This

combination of simulated processing and querying hypothetical logs would provide a powerful tool to conduct "what-if" analysis on laboratory or process changes.

6. Current Status and Future Plans

We are just beginning the implementation of the applications. Our plan is to implement the WIP interpreter first since it requires the fewest features from OBJFADS and POSTGRES. The initial implementation will be done using commercial INGRES. We plan to install the WIP interpreter in the MICROLAB if the performance is acceptable.

This summer (1987) we will implement the data types required by both applications for POSTGRES starting with the geometric data types. After these are completed we will build a prototype version of the facility editor to test the performance of the system.

The most significant research question is whether OBJFADS and POSTGRES will be fast enough to make this approach to implementing CIM applications practical. We believe they have the right features, but now we must prove it!

Acknowledgements

We want to thank Dave Hodges who has created the environment that allows us to pursue this work. Without his direction we would never have gotten access to the people with the expertise in semiconductor manufacturing that guided this work.

References

1. *Inside MAC*, Apple Computer, Inc., Cupertino, CA, 1985.
2. M. Astrahan and et.al., "System R: A Relational Approach to Data", *ACM TOD*, June 1976.
3. J. Banerjee and et.al., "Multimedia Information Management in an Object-Oriented Database System", DB-046-87, MCC, February 1987.

4. D. Bobrow and G. Kiczales, "Common Lisp Object System Specification", Draft X3 Document 87-001, Am. Nat. Stand. Inst., February 1987.

5. O. P. Buneman and E. K. Clemons, "Efficiently Monitoring Relational Databases", *ACM Trans. Database Systems*, Sep. 1979, 368-382.

6. M. Carey and D. DeWitt, "Extensible Database Systems", *Proc. Islamorada Conference on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985.

7. U. Dayal and et.al., "A Knowledge-Oriented Database Management System", *Proc. Islamorada Conference on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985.

8. C. Y. Fu and et.al., "Expert System for IC Computer-Aided Manufacturing", To appear *Proc. SEMICON EAST*, September 1987.

9. J. Gettys, "Problems Implementing Window Systems in UNIX", *Proc. Winter USENIX Technical Conf.*, Jan. 1986, 89-97.

10. J. Gray and M. Anderton, "Distributed Databases -- Four Case Studies", Tandem Technical Report 85.5, Tandem Computers, Inc., June 1985.

11. T. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proc. 1984 ACM-SIGMOD Conf.*, Boston, MA, May 1984.

12. D. A. Hodges and L. A. Rowe, "Information Management for CIM", *Proc. Adv. Res. in VLSI*, Palo Alto, CA, Mar. 1987.

13. *IMS/VS Message Format Services User's Guide*, IBM Form No. SH20-9053, Armonk, NY, 1980.

14. D. Maier and J. Stein, "Development of an Object-Oriented DBMS", *Proc. 1986 ACM OOPSLA Conf.*, Portland, OR, Sep. 1986.

15. F. Maryanski and et.al., "The Data Model Compiler: a Tool for Generating Object-Oriented Database Systems", Unpublished manuscript, Elect. Eng. Comp. Sci. Dept., Univ. of Connecticut, 1987.

16. J. Nievergelt and et.al., "The Grid File: An Adaptable Symmetric Multikey File Structure", *ACM Trans. Database Systems 9*, 1 (March 1984).

17. L. A. Rowe, "A Shared Object Hierarchy", *Proc. Int. Wkshp on Object-Oriented Database Systems* , Asilomar, CA , Sep. 1986.

18. L. Rowe, "Object FADS Project Status Report", Memo No. M87/20, Electronics Research Laboratory, U.C. Berkeley, April 1987.

19. L. A. Rowe and M. R. Stonebraker, "The POSTGRES Data Model", Submitted for publication, Mar. 1987.

20. L. A. Rowe and C. B. Williams, "Berkeley Process-Flow Language", In preparation, May 1987.

21. P. Schwarz and et.al., "Extensibility in the Starburst Database System", *Proc. Int. Wkshp on Object-Oriented Database Systems* , Asilomar, CA , Sep. 1986.

22. J. Starkey, Personal communication, 1987.

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

24. M. R. Stonebraker, "Object Management in POSTGRES Using Procedures", *Proc. Int. Wkshp on Object-Oriented Database Systems* , Asilomar, CA , Sep. 1986.

25. M. R. Stonebraker and L. A. Rowe, "The Design of POSTGRES", *Proc. 1986 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1986.

26. M. Stonebraker, "POSTGRES Storage System", Submitted for publication, 1987.

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

28. "Tandem 16 Pathway Reference Manual", 82041, Tandem Computers Inc., Feb. 1980.

29. *INGRES/STAR*, Version 1.0, Relational Technology, Inc., Berkeley, CA, Jan. 1987.

30. *INGRES Reference Manual*, Version 3.0, VAX/VMS, Relational Technology, Inc., Berkeley, CA, May 1984.

31. D. Wolfson, Personal communication, 1987.