Copyright © 1988, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

THE DESIGN AND IMPLEMENTATION OF THE BERKELEY PROCESS-FLOW LANGUAGE INTERPRETER

by

Christopher B. Williams

Memorandum No. UCB/ERL M88/72

15 November 1988

COVER PAGE

THE DESIGN AND IMPLEMENTATION OF THE BERKELEY PROCESS-FLOW LANGUAGE INTERPRETER

by

Christopher B. Williams

Memorandum No. UCB/ERL M88/72

15 November 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

THE DESIGN AND IMPLEMENTATION OF THE BERKELEY PROCESS-FLOW LANGUAGE INTERPRETER

by

Christopher B. Williams

Memorandum No. UCB/ERL M88/72

15 November 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CHAPTER 1

Introduction

This report describes the design and implementation of a generic interpreter for specifications written in a language that describes a recipe to fabricate integrated circuits (IC's). The generic interpreter is used to produce application-specific interpreters that can produce input to various process simulators or control the operation of a fabrication facility. The remainder of this introduction describes our model of processes and views, problems with current ways of specifying processes, the approach we are pursuing to solve these problems, and comparisons with other process specification languages.

1.1. Processes and Views

Manufacturing semiconductor devices is a complex and lengthy task that can involve well over a hundred operations and a period of weeks. The devices are built on and out of wafers of silicon by processing them in specialized equipment. Each equipment operation is called a *step*, and the sequence of steps that produces a working device is called a *process*. We are concerned with the description of a process and, in particular, making practical use of a formal specification of a process.

At first glance, the description of a process is straightforward: specify each step and then specify the sequence of steps that forms a process. However, processes are viewed differently by different people, such as technicians and engineers. While semiconductor processing equipment is technologically advanced, from the technician's hands-on point of view it differs little from a tool in a machine shop. A technician's view of a process step describes how to set up and operate the processing equipment by pushing buttons, moving materials, and turning knobs.[1] To the technician's unaided eye, the only apparent change to the wafers during processing is at most a change in the color and pattern on the wafer. In an engineer's view of a process step, the equipment operation is summarized by the chemical and physical environment that the equipment creates around the wafer. To the engineer, the changes to the wafer are on the scale of micrometers.

[1]For the purposes of this comparison, a knob is any setting in the machine, including a variable in a controlling computer.

Thus, a process step can be described either by the equipment operation or by the results it produces. Each description is called a *view* of the process step, and each type of view is based on a particular frame of reference, such as chemistry or equipment operation. Each frame of reference is made up of *models*, such as reaction rates and crystals in the chemistry frame of reference, or knob calibration and manual dexterity in the equipment operation frame of reference. A view describes the process in terms of the models of a particular frame of reference.

Since the typical manufacturing process is too large to be easily comprehended as a whole, views and their frames of reference are used as tools for understanding and controlling the process. People generally work with views of the process that simplify each step or restrict attention to a subset of the steps. One common simplification is reducing a sequence of steps a single *abstract* step when the frame of reference groups several equipment steps together in one simpler model. For example, the steps coat, bake, expose, and develop can be reduced to the abstract step of patterning resist. In views that aid the understanding of a process, the abstract steps generally assign a goal or intent to sequences of equipment steps. In views used to simulate a process, the abstract steps represent the aggregation of equipment steps into one step in the simulation model.

1.2. Management of Views

In practice there is no comprehensive process description, only a collection of process views. The frames of reference underlying the different views can have some models in common, such as the concept of heat. Thus, the views may contain redundant information, since each must describe the use of the model (e.g., specify a temperature). Since the views may contain duplicate information, they must be carefully managed to keep them consistent with each other.

Several common view types have emerged in the semiconductor industry, with some being more formally defined than others. Two such views, as already noted, are the description of process steps in terms of the chemical and physical changes that the equipment causes in a silicon wafer and in terms of the operation of the equipment. In addition, factory managers view the process as an ordering on all work currently in progress and each step as a utilization of equipment. Data relevant to the manager's view includes how long each step takes and how lots are split for rework.

A document called a *run-sheet* is the traditional description of a manufacturing process. The run-sheet describes the steps to be done and the choices to be made during a process. It often describes goals in terms of physical changes to the wafers and summaries of equipment steps (e.g., "Grow 200 Å of oxide in wet O_2 for 20 minutes"). The run-sheet specifies criteria for decisions, such as when to scrap wafers and how to adjust a step to accommodate variations in previous steps. However, most of the information is specified in English or is structured for interpretation by a human. Similar information is found in engineers' notebooks.

In most modern facilities, a supplementary process view, used for day-to-day operations, is managed by a computer system that supports the definition of individual steps and the definition of processes from sequences of steps [1] [2]. The system typically supports abstract steps in the factory manager's view of a process, where the usual form of a step is to prepare the wafers (e.g., clean and inspect), process them, and then measure some results. Such "process supervision" systems are an improvement over textual run-sheets, but much of the information inside the boundaries of the steps is still intended for humans.

Engineers and managers use simulators as an aid in the design and manufacture of devices, because computer simulation allows relatively inexpensive experimentation [3] [4] [5]. Some simulators model the creation of devices and their behavior while others model the movement of material in a factory. Input data for a simulation includes a specification of the manufacturing process. The simulators each have different models for doing their computations, and have different input formats, even where the models overlap. Thus, each type of simulation requires the generation of another view of a process.

Taken together, the views provided by the run-sheet, the process supervision system, simulators, and engineers' notebooks define the manufacturing process. Most people use only one view. Consequently, the different specifications maintained for the different views can be inconsistent. The views must be kept consistent where they overlap or else conflicting statements can be made about the process. Also, inconsistent views can make diagnosis of problems difficult when the diagnosis needs to consider the interactions of several views at once. When views do not agree, there is the additional problem of determining which one is more accurate or appropriate. In general, inconsistent views lead to inefficiency in process design, IC manufacturing, and factory management.

1.3. BPFL

Our solution to the problem of coordinating views of a process is to create a single language in which to describe the device manufacturing process. The Berkeley Process-Flow Language (BPFL) provides a comprehensive frame of reference for describing processes by using models that can be automatically transformed to the models of other views [6]. A process described in BPFL can be used both to manufacture an IC and to provide input to simulators.

Since a semiconductor manufacturing process consists primarily of sequences of operations, with occasional tests and repetitions, it is easy to think of the factory as a machine to be programmed to produce the desired devices. Consequently, a language for describing processes has many of the same facilities as a programming language for a computer. But a factory manipulates materials as well as information, so a programming language has to be extended to accommodate the physical world. BPFL was designed around this philosophy. The programming language upon which BPFL is based is Common Lisp [7]. We chose Common Lisp because it is easily extended. Consequently, we can rapidly prototype new constructs in BPFL. The ultimate implementation of BPFL does not have to be in Lisp.

Since the BPFL frame of reference encompasses the models of several views, a process description in BPFL will contain information needed by many different applications, such as different simulators or a process supervision system. The desired information is extracted from the process description by an application-specific interpreter. The interpreters provide a mechanism for keeping multiple views consistent by deriving them from a common source.

In the current implementation of BPFL interpreters, a *core interpreter* carries out the basic functions of parsing and translation of a process description written in BPFL. This core interpreter may be extended to create *application-specific interpreters* that extract the information necessary for a specific view. Two application-specific interpreters have been implemented that generate input files for the simulators SIMPL-2 [4] and SUPREM-3 [5].

1.4. Related Work

BPFL is not the first attempt to formalize process descriptions. The Fable project aimed to solve the problem of managing process descriptions, which at the time were almost all in run-sheet form [8]. The approach taken in Fable was to use sophisticated programming language constructs to model the behavior of equipment and the factory during process steps (e.g., complex data types, tasking, and exception handling). The models were focussed on the equipment-operation view of a process. In addition, procedures could be constructed from the equipment models to provide manageable abstract steps from which the entire process could be defined. The procedures were to be executed by an interpreter that would keep the procedural model of the process synchronized with the physical facility. The project did not address the problem of maintaining multiple views, such as simulation views.

A project at MIT provides for abstract steps similar to the industrial process supervision systems, but the operations contain additional information for use by simulators. The process descriptions are interpreted to give the desired results for manufacturing or simulation. In this approach, the simulation views are pre-computed and little interpretation is done on them. The interpreter acts to connect the steps of the process.

1.5. Project Overview

Our approach to process description and interpretation is similar to the MIT approach. The difference is that we do not precompute the simulator view and instead write interpreters to construct the simulator input. Using BPFL we have specified several processes. We have implemented a generic interpreter and two application-specific interpreters. A work-in-progress interpreter that will supervise equipment operations in a fabrication facility is under development.

The remainder of this report describes BPFL and the interpreters. The next chapter gives some examples of the description of a process with BPFL and the results of interpreting it. Chapter 3 discusses the implementation of the core and application interpreters built so far. Chapter 4 describes our experience working with BPFL and the interpreters, and Chapter 5 discusses some future possibilities for the language.

CHAPTER 2

Examples of BPFL and Interpreters

This chapter provides some examples of a process described with BPFL and some views of the process. The process, which specifies how to manufacture an N-well CMOS device, was developed at the Berkeley Microfabrication Laboratory. The views of the process are the input files for the simulators SIMPL-2 and SUPREM-3, which model cross sections of a wafer. The views were derived from the process description by application-specific interpreters.

2.1. An Overview of BPFL

As mentioned in the previous chapter, BPFL is based on Common Lisp [2], although there are several restrictions in BPFL to make the prototype interpreters easier to implement. Most constructs in BPFL are similar to the corresponding ones in Lisp.

A process or abstract step is represented by a *function call* in BPFL. The implementations of the abstract steps themselves are represented by *function* definitions, which contain a sequence of function calls for the component steps. An entire manufacturing process is considered to be one abstract step and has a corresponding top-level function. As with Lisp, BPFL functions and calls to those functions are represented as lists of items between parentheses. Literal values are preceded by a single quote (') and *keyword* symbols are preceded by a colon (:). Keyword symbols are used as constants and to name arguments in function calls.

A function call is a list of items. The first is the *name* of the function call and the remainder are the *actual arguments* to the function being called. Arguments can be either positional (i.e., given in a specified order) or keyword arguments (i.e., preceded by a keyword that names the argument). When a function call is evaluated, the actual arguments are evaluated and their values are passed to the function. For example, the function call

```
(+ z 10)
```

will return the value of `z` plus 10. Function calls are also known as *forms* in Lisp terminology.

Some functions are called *special forms* (e.g., `if`, `progn`, or `while`). These forms do not evaluate their arguments in the usual fashion. Instead, they evaluate them individually, and depending on the results, may evaluate some arguments several times, or not at all. For example, the `if` special form evaluates its first "argument" and then evaluate either the "then" argument or the "else" argument. For example, the form

```
(if (> z 0) z 0)
```

evaluates to zero if `z` is not positive, and otherwise returns the value of `z`. The `setf` special form does not evaluate its first argument, but instead uses it to identify a location in which to store the second argument.

Functions are categorized into *built-in* functions and *user-defined* functions. BPFL has a variety of built-in functions, taken from Common Lisp, to specify arithmetic, data access, and control flow. User-defined functions specify the construction of a process.

A function is defined at one of three abstraction *levels*: the *flow* level, the *generic-equipment* level, or the *equipment-specific* level. A function is defined with one of the symbols `defflow`, `defgeneric`, or `defspecific` to show its level of abstraction. Each level uses a different set of models. A flow level function describes material changes in the wafers being processed. A generic equipment level function describes the chemical and physical environment to which wafers are exposed. An equipment specific level function describes how a piece of equipment is to be set up and run. Functions defined at the flow level are generally implemented in terms of functions at the generic equipment level, which are in turn implemented by functions at the equipment specific level. Functions may also be implemented in terms of functions at the same level. Using a hierarchy of functions, a process described with BPFL can state what is to happen or be changed, and how it will be accomplished.

The flow and generic-equipment levels have *primitive* functions that define the basic operations and models for the level. For example, some primitives at the flow level are "deposit a layer of material," "implant ions," or "remove a layer of material." Primitives at the generic-equipment level describe the chemical reaction environments provided by major classes of equipment, such as furnaces, plasma systems,

or wet sinks. Primitives do not have implementations as BPFL functions and only appear as the names of function calls.

Interpreters use calls to primitive functions to extract information from one abstraction level without having to see how the level is implemented. If an application interpreter uses a primitive function, it is considered built-in to that interpreter. In Lisp the name of the function call always determines the function to be called, but in BPFL that name may be overridden by an `:implemented-by` argument. Thus, if a primitive is built-in to an application, it will be recognized. Otherwise, the function call will be defined via the `:implemented-by` argument.

The BPFL abstraction levels define some common process views used by engineers to describe processes to each other. A given application will not use a BPFL abstraction level directly, because each application has a different frame of reference. Nevertheless, describing a process in terms of these levels is a great aid to capturing information about the process. Within the BPFL framework, additional information can always be provided to accommodate an application. This amounts to adding models to the BPFL frame of reference. Because BPFL provides a common frame of reference, the addition does not become a disjoint process view with the attendant problem of inconsistency.

2.2. BPFL and Simulator Examples

Figure 2.1 shows the top-level function for the Berkeley N-well CMOS process. The function is displayed in the machine-readable syntax used by the BPFL system. A structured editor has beed developed that simplifies entry and correction of BPFL functions [9]. In addition, we are experimenting with other user-interfaces and language syntaxes to specify a BPFL function.

The function begins by declaring two arguments. The argument `analog`, which defaults to false, will be tested later in the function to determine whether steps for analog device options should be included. The argument `lot-size`, which defaults to 25, will determine the number of wafers to be processed. The function next allocates wafers (the `allocate-lot` function call), sets some global parameters (`*def-resist*` and `*def-resist-thickness*`), and then, starting with `CMOS-Well-Formation`, calls other BPFL functions to do the major sub-sections of the process. The allocated wafers are grouped into named *lots* (e.g., `MAIN`) so that the process description is not bound to one exact format

```
(defflow CMOS-Nwell ((analog nil) (lot-size 25))
  "3 um, N-well, single poly-Si, single metal"
  (declare (special *def-resist* *def-resist-thickness*))

  ; Allocate wafers.
  (allocate-lot :size lot-size
                :material '(silicon (type p)
                                    ;#u unit value
                                    (resistivity #i(#u(18 ohm-cm) #u(22 ohm-cm)))
                                    (crystal <100>))
                :thickness #u(2 mm)
                :names (list (list 'MAIN (make-interval 1 (- lot-size 3)))
                             (list 'WELL (- lot-size 2))
                             (list 'NCH (- lot-size 1))
                             (list 'PSG lot-size))
                )

  ; Define global variables
  (setf *def-resist* '(photo-resist (polarity positive) (brand kodak-820)))
  (setf *def-resist-thickness* #u(1.2 um))

  ; Abstract steps
  (CMOS-Well-Formation :lot '(MAIN WELL))
  (CMOS-Isolation-Formation :lot '(MAIN WELL))
  (CMOS-Threshold-Adjustment :lot '(MAIN WELL))
  (if analog
      (CMOS-Analog-Options) :lot '(MAIN WELL))
  (CMOS-Gate-Poly-Formation :lot '(MAIN WELL NCH))
  (CMOS-Source-Drain-Formation :lot '(MAIN WELL NCH))
  (CMOS-Passivation-Metalization :lot '(MAIN WELL NCH PSG)))
```

Figure 2.1 – BPFL function for the Berkeley CMOS process.

of wafer handling. As each major sub-section is called, it is restricted to operate on a subset of the wafers via the `:lot` keyword argument. When this top-level function returns, the process is completed.

Figure 2.2 shows the specification of the first major subsection of the N-well process, `CMOS-Well-Formation`. All the functions called are primitives for the flow level of BPFL. Each function defines a major change to the wafer (e.g., `deposit`) or a support operation (e.g., `measure`). In addition, the `:implemented-by` argument specifies the function that implements the primitive in terms of generic equipment steps.

Figure 2.3 shows the function `CMOS-Init-Oxide`, which implements the "initial oxidation" step of `CMOS-Well-Formation`. It is defined at the generic-equipment level of BPFL and uses the generic-equipment primitive `furnace` to describe the processing of the wafer. This primitive is implemented in terms of the equipment-specific function `CMOS-Init-Oxide-Furnace` (not shown) that will allocate a furnace tube and cause the appropriate recipe to be loaded. `CMOS-Init-Oxide`

```
(defflow CMOS-Well-Formation ()
  (declare (special *def-resist* *def-resist-thickness*))

  (grow :doc "Initial Oxidation"
        :material 'silicon-oxide
        :thickness #u(1000 A)
        :implemented-by 'CMOS-Init-Oxide)

  (measure :attribute 'silicon-oxide-thickness
           :lot 'WELL)

  (mask :mask 'NWEL
        :polarity :negative
        :thickness *def-resist-thickness*
        :lot 'MAIN)

  (implant :doc "Well Implant"
           :dopant 'phosphorus
           :dose #u(4E12 /cm^2)
           :depth #u(0.19 um))

  (other :doc "TCA clean"
         :lot nil)

  (etch :material 'silicon-oxide
        :thickness '(all #u(1000 A)))

  (CMOS-Etch-Resist)

  (drive-in :doc "Well Drive-in"
            :junction-depth #u(3 um)
            :oxide-thickness #u(3000 A))

  (measure :attribute 'silicon-oxide-thickness :lot 'WELL)
  (measure :attribute 'junction-depth :lot 'WELL)
  )
```

Figure 2.2 – BPFL function for well formation.

```
(defgeneric CMOS-Init-Oxide ()
  (furnace :sequence '((temp #u(1000 c))
                       (gas oxygen on) (wait #u(5 min))
                       (gas steam on) (wait #u(11 min))
                       (gas oxygen on) (wait #u(5 min))
                       (gas nitrogen on) (wait #u(20 min)))
           :implemented-by 'CMOS-Init-Oxide-Furnace))
```

Figure 2.3 – BPFL function for initial oxidation.

could also have included other steps, such as a cleaning operation, to be part of the abstract initial oxidation step.

```
* Initial Oxidation *
WHICH PROCESS ? OXID
OXIDE THICKNESS (micro-meter) ? 0.1
Xt (micro-meter) ? 0.1
Xe (micro-meter) ? 0.05
u1 ? 0.1
u2 ? 0.5
u3 ? 0.9
d1 ? 0.1
d2 ? 0.5
d3 ? 0.9
DO YOU WANT TO DRAW THE CROSS SECTION (yes or no) ? yes
WHICH PROCESS ? DEPO
NAME OF THE MATERIAL ? RST
THICKNESS OF THE MATERIAL (micro-meter) ? 1.2
ISOTROPIC, ANISOTROPIC, OR VERTICAL (I, A, or V) ? V
DO YOU WANT TO DRAW THE CROSS SECTION (yes or no) ? no
WHICH PROCESS ? EXPO
WHICH MASK ? NWEL
INVERT THE MASK (yes or no) ? yes
NAME OF THE EXPOSED RESIST ? ERST
DO YOU WANT TO DRAW THE CROSS SECTION (yes or no) ? no
WHICH PROCESS ? DEVL
NAME OF THE LAYER TO BE DEVELOPED ? ERST
DO YOU WANT TO DRAW THE CROSS SECTION (yes or no) ? yes
* Well Implant *
WHICH PROCESS ? IMPL
IMPLANTATION TYPE (p or n) ? N
DOSE (cm**(-2)) ? 4.0e+12
STANDARD DEVIATION (micro-meter) ? 0.6
PEAK DEPTH (micro-meter) ? 1.5
BLOCK THCKNESS (micro-meter) ? 0.285
DO YOU WANT TO DRAW THE CROSS SECTION (yes or no) ? yes
WHICH PROCESS ? ETCH
WHICH LAYER DO YOU WANT ETCH ? OXID
ETCH ALL (yes or no) ? no
AMOUNT OF VERTICAL ETCH (micro_meter) ? 0.101
RATIO X/Z OF ETCHING (0.0 <= RATIO <= 1.0) ? 0.5
DO YOU WANT TO DRAW THE CROSS SECTION (yes or no) ? yes
WHICH PROCESS ? ETCH
WHICH LAYER DO YOU WANT ETCH ? RST
ETCH ALL (yes or no) ? no
AMOUNT OF VERTICAL ETCH (micro_meter) ? 1.212
RATIO X/Z OF ETCHING (0.0 <= RATIO <= 1.0) ? 0.5
DO YOU WANT TO DRAW THE CROSS SECTION (yes or no) ? yes
* Well Drive-in *
WHICH PROCESS ? OXID
OXIDE THICKNESS (micro-meter) ? 0.3
Xt (micro-meter) ? 0.3
Xe (micro-meter) ? 0.15
u1 ? 0.1
u2 ? 0.5
u3 ? 0.9
d1 ? 0.1
d2 ? 0.5
d3 ? 0.9
DO YOU WANT TO DRAW THE CROSS SECTION (yes or no) ? yes
```

Figure 2.4 – The SIMPL view of CMOS process.

The N-well process can be modeled by several simulators if the appropriate input formats are given. Figure 2.4 shows the input to the SIMPL-2 simulator for the N-well process through the end of the well formation. This format was created by an interpreter that traversed the BPFL description of the N-well process in Figure 2.2 and noted the occurrence of primitives at the flow level. The information from the primitives was used to create the SIMPL-2 view of the process. Figure 2.5 shows the SUPREM simulator view of the well formation. This view was created with a different interpreter that searched for primitives at the generic-equipment level. The text comments have been transferred from the BPFL description to the simulator inputs to improve the readability of the files.

If the application interpreters were built into the simulators, the simulators could be thought of as accepting BPFL process descriptions as input. The advantage of using one process description is that it allows automatic checking between the different process views that can avoid errors introduced by inconsistent specifications.

```
TITLE **** SUPREM translation of BPFL function cmos-nwell ***
$ Wafer index: 1
INITIALIZE SILICON <100> THICKNES=2000.0001
$ Doc: "Initial Oxidation"
DIFFUSION TIME=5 TEMPERAT=1000 DRYO2 PRESSURE=1
DIFFUSION TIME=11 TEMPERAT=1000 WETO2 PRESSURE=1
DIFFUSION TIME=5 TEMPERAT=1000 DRYO2 PRESSURE=1
DIFFUSION TIME=20 TEMPERAT=1000 NITROGEN PRESSURE=1
$ Doc: "Well Implant"
IMPLANT DOSE=4.0e+12 ENERGY=150 PHOSPHOR PEARSON
ETCH OXIDE ALL
$ Doc: "Well Drive-in"
DIFFUSION TIME=240 TEMPERAT=1150 DRYO2 PRESSURE=1
DIFFUSION TIME=240 TEMPERAT=1150 NITROGEN PRESSURE=1
```

Figure 2.5 – SUPREM view of CMOS process.

CHAPTER 3

Design and Implementation of the Interpreters

This chapter describes the implementation of the interpreters for BPFL. The core interpreter "executes" a process description written in BPFL and provides hooks for application-specific interpreters to extract information from the execution. The application-specific interpreters use the extracted information to generate their output. The BPFL examples and simulator views from Chapter 2 will be used to illustrate the discussion.

3.1. Design of the Core Interpreter

The core interpreter is built around the Lisp concept of *evaluation*. All expressions in BPFL can be evaluated to yield a data value. Sometimes the evaluation result is the same as the input (i.e., a constant value), as happens with numbers, strings, and keywords. Symbols are treated as the names of variables and evaluate to the value stored in the variable. Lists are treated as function calls and evaluate to whatever the called function returns.

An interpreter makes use of a process description by evaluating the top-level function for the process and monitoring the side-effects of the evaluation. Each time the core interpreter evaluates an expression, it creates a data structure called a *frame* to control that evaluation. The frame is discarded when the evaluation is completed. When evaluating a function call, the body of the function must be evaluated to determine the result of the call, creating a subordinate frame. Subordinate frames are also created to determine the actual arguments to a function and while evaluating special forms. Consequently, the interpreter maintains a *stack* of frames. The top of the stack is the frame for the expression most recently evaluated that has not yet returned a result. This frame is called the *current frame*. The bottom of the stack is the frame evaluating the top-level function of the process.

Each evaluation is associated with a *lot*, which is a set of wafers that are to be processed together. A lot may have a name and a wafer may belong to more than one lot at a time. Each wafer has a logical wafer number (starting from 1) for use in BPFL functions.

A lot of wafers under the control of a process is a *run*. In the interpreter, each BPFL run is managed by a `run` data structure that contains information about the wafers in the run, the values of all the variables, and the frame stack. The wafer information is in two parts. One part is the mapping between logical wafer numbers and the data structures describing physical wafers. The second part is the mapping from lot names to the logical wafer indexes of the member wafers. For example, the `MAIN` and `WELL` lots contain wafers 1 through 22 and 23, respectively. Each wafer is known to be positively doped. The lot management functions defined in BPFL control these mappings.

There are three types of variables in a `run`: *global* (i.e., those declared `special`), *local* (i.e., formal arguments and `&aux` variables), and *actual arguments* (i.e., the positional and keyword arguments passed to the function). The function evaluation mechanism and the `setf` special form manage the values of variables. Technically, the interpreter can execute more than one run at a time by maintaining multiple `run` structures and frame stacks, although no application has yet required this facility.

In the core interpreter, the frame data structure is called an `eval-frame`. It specifies the function in the interpreter, called the *dispatch* function, that is currently handling the evaluation represented by the frame, and may contain some state information. An `eval-frame` also records how the return value of the evaluation will be handled and which frame is interested in the result. The usual state information for an evaluation is the code fragment being evaluated plus an index into the code fragment if it has several parts. The index, called a *code-position*, is a list indicating how many list items should be skipped to reach the one of interest. For example, in the list `(A B (C D))`, the symbol `B` is indexed by "`(1)`," and the symbol `D` is indexed by "`(2 1)`." The code position is not a direct Lisp pointer because it must be possible to move backwards in a list. The code-position is primarily used when evaluating special forms and function calls.

When a function call is evaluated, a special type of `eval-frame`, called a `stack-frame`, is created in addition to the usual `eval-frame`. A `stack-frame` will control the evaluation of the function to be called and has additional state variables, such as an identifier used to tag the local variables. The `stack-frame` is used to store the actual arguments of the function call. It will also store a list of variables declared `special` and the composition of the `current` lot for the function. Once the

`eval-frame` for the function call has evaluated and stored the actual arguments, control is passed to the `stack-frame` to evaluate the function body. The dispatch function for the `stack-frame` will assign actual arguments to formal arguments, initialize variables, and then evaluate of the body of the function. The `stack-frame` serves as a marker in the frame stack for finding values of variables.

3.2. The Main Loop

The basic operation of the core interpreter is to create a run and execute the evaluation main loop. Run creation requires initializing a `run` structure, fetching the top-level BPFL function, and creating the first `stack-frame` to evaluate it. The main loop iterates over the following algorithm:

- The run is checked for a termination condition, such as an error flag or a lack of more code to execute.
- If the run is still active, the dispatch function for the *current frame* is called.
- The dispatch function will do some work and leave the frame stack in a consistent state.

The functionality of the core interpreter is contained in the frame dispatch functions.

A dispatch function either manipulates the current frame, invokes a subordinate evaluation, or returns a value. More specifically, a dispatch function leaves the frame stack consistent either by advancing its own state stored in the current frame, by arranging for a new dispatch function to manage the current frame, by requesting an evaluation that pushes a new frame onto the frame stack, or by causing the current frame to return a value. When requesting an evaluation, a dispatch function can request that control be returned to itself, or that the result of the evaluation be used as the return value for the current evaluation as well. When returning a value, the frame that originally requested the value will become the current frame, and all the intervening frames are discarded.

The dispatch functions for function calls and special forms use the code-position as an instruction pointer or program counter. For example, the `if` form consists of the name, a conditional expression, a "then" clause, and an optional "else" clause. When a dispatch function needs to evaluate a data value, it sets its code-position to the item and invokes an `eval-frame`. When a result is returned, the code-position is advanced by one before the next iteration of the main loop. If the dispatch function for the `if` special form finds its code position pointing to the "then" clause, this means that the evaluation of the conditional clause must have just returned. When evaluating the arguments to a function call, more state

information is needed to record whether the argument denoted by the current code-position is positional or keyword.

Figure 3.1 shows the tail of the frame stack during the evaluation of the `:size` keyword argument to the following function call taken from figure 2.1:

```
(allocate-lot :size lot-size
              :material '(silicon (type p)
                                  (resistivity #i(#u(18 ohm-cm)
                                                    #u(22 ohm-cm)))
                                  (crystal <100>))
              :thickness #u(2 mm)
              :names (list (list 'MAIN (make-interval 1 (- lot-size
                                                           3)))
                           (list 'WELL (- lot-size 2))
                           (list 'NCH (- lot-size 1))
                           (list 'PSG lot-size))
              )
```

The frame marked F is performing the function call and has created a frame, marked V, to evaluate the first keyword argument. The V frame is the current frame and its dispatch function, `eval-variable`, will generate a value by looking up the variable named `lot-size`. The value will be returned to the `returned-value` slot of the F frame and the V frame will be removed from the stack. During following iteration of the main loop, the F frame will be the current frame and will save the actual argument value for the eventual function call. It will then evaluate the `:material` argument.

Figure 3.2 shows a similar example of the evaluation stack. Here, the frame for the `if` statement of figure 2.1 has just become the current frame again after requesting the value of the `analog` variable. The

Frame V	(current frame)
dispatch:	`eval-variable`
code:	`lot-size`
return-frame:	F

Frame F	
dispatch:	`eval-funcall`
state:	`keyword arguments`
code:	`(allocate-lot :size lot-size :material ...)`
code-posn:	(2) (the third item of the code slot)
returned-value:	uninitialized at this point

Figure 3.1 – Evaluating a variable as an argument to a function call.

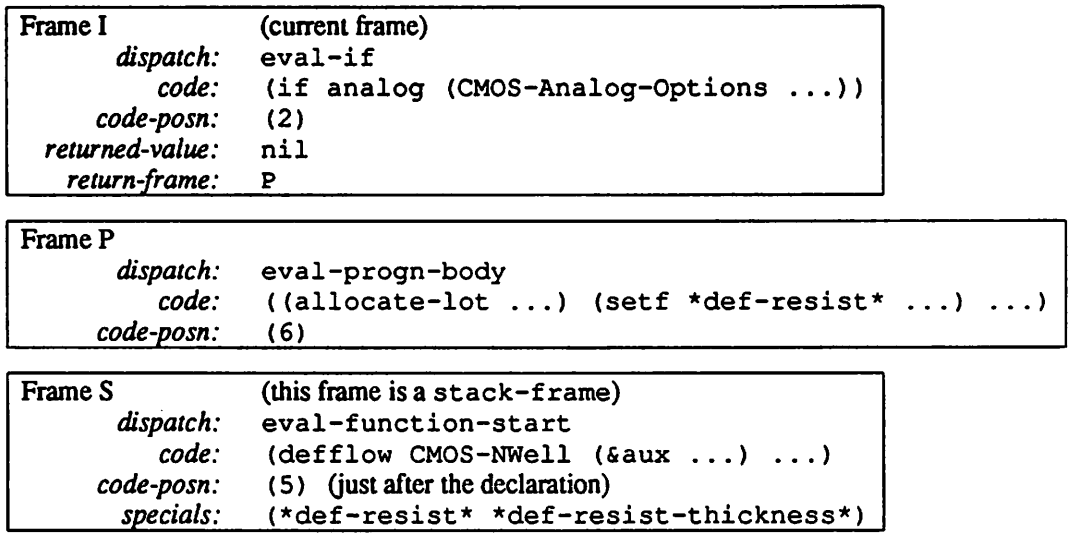

Figure 3.2 – State of the stack after an evaluation.

returned value was `nil`, so the `if` special form will execute the "else" portion of the list. In this example, there is nothing to do, so the frame will return `nil` and will be popped from the stack. The `code-posn` of the P frame is advanced, and the interpreter loop will continue. If there had been an "else" clause, the I frame would have created another `eval-frame` and initialized it to return its value to the P frame, which is evaluating the body of the `CMOS-NWell` function. Note that figure 3.2 contains the entire stack for the run, including the `stack-frame` for the top-level function `CMOS-NWell`. The P frame was created by the S frame to interpret the function body after the local variables had been initialized and the `special` declarations had been recorded.

3.3. Application interface to the core interpreter

Since BPFL has a syntax and data types similar to Lisp, the core interpreter is written in Common Lisp. The advantage of using Lisp instead of a block structured language (e.g., Pascal or C) is that the list data type and many of the functions built into BPFL do not have to be implemented from scratch. For convenience, interpreters are also written in Lisp. The application-specific interpreter uses a library of functions to interface to the core interpreter. The application-specific interpreters developed so far operate in two passes – one that interprets the process via the core interpreter main loop and one that generates the application-specific output. The BPFL function-call mechanism of the core interpreter provides the ability

for interpreters to trace the execution of a process description. First, the interpreter specifies the BPFL functions that it is interested in by registering them as *built-in* with the core interpreter. When the core interpreter evaluates a function call to a built-in function, it calls a Lisp function in the interpreter.

At this point, the interpreter can examine the actual arguments of the function call, usually storing them for later analysis. It can then cause the BPFL function call to return a value or allow the core interpreter to continue with the function call as it normally would, using the function name or the `:implemented-by` argument. In this way, the application-specific interpreter can control how deep and detailed the evaluation of the process description will be. Note that the interpreter is isolated from the implementation of the core interpreter. The application-specific interpreter always deals with the current frame, so it only needs to request values from that frame.

For example, in figure 2.2 the first function call to be evaluated is the `grow` primitive. The values of the four actual arguments will be saved in a core interpreter `stack-frame` and the function to be called will be determined. In the SIMPL interpreter, the `grow` function has been registered as being built-in. The core interpreter calls a function in the SIMPL interpreter (specified when `grow` was registered) that will save the `:material` and `:thickness` arguments from the call. The SIMPL interpreter then indicates that the `grow` function should simply return, because the SIMPL interpreter operates only at the BPFL flow level. The `:material` and `:thickness` arguments are used in the second pass of the SIMPL interpreter to generate an `OXID` statement.

At the same `grow` function call, the SUPREM interpreter des not interrupt the core interpreter execution. The core interpreter will note that the `:implemented-by` argument has been given, retrieve the BPFL function `CMOS-Init-Oxide` (shown in figure 2.3), and continue interpretation. The evaluation of the `CMOS-Init-Oxide` function will evaluate the call to the generic equipment level primitive `furnace`. Just as the `grow` primitive was registered by the SIMPL interpreter, the SUPREM interpreter traps the call to `furnace`. The second pass of the SUPREM interpreter examines the `:sequence` argument to the `furnace` call and generates `DIFFUSION` statements accordingly.

After the top-level function for the entire process has been evaluated, the application-specific interpreters have recorded the arguments to a sequence of "interesting" function calls. The interpreters

can then generate views based on the primitives called. Sometimes the interpreters must transform the BPFL primitives to the models of the target interpreter. For example, SIMPL-2 does not have a diffusion model. Thus, the `drive-in` primitive must be approximated by making the previous implant deeper. This implementation is easily accomplished in the interpreter by scanning ahead during the second pass. No output for an implantation will be created until all drive-in steps have been accounted for.

Similarly, the SUPREM-3 interpreter must try to determine the initial doping of the silicon substrate. In the examples of Chapter 2, the doping was given in terms of surface resistivity, and no specific dopant was requested in the call to `allocate-lot`. Since the SUPREM model requires a specific element to determine the doping, the `INITIALIZE` statement does not indicate initial doping. The solution is to provide the doping information needed for SUPREM, or the interpreter may assume that the dopant is boron.

3.4. Other Implementation Details

The BPFL specification motivates the use of several data structures in the core interpreter. The simplest are the structures for the unit and interval BPFL data types. These are handled by the "#" read-macro facility in Lisp. The difficulty was in providing the functionality required of the math functions, such as adding intervals of unit values. Units and intervals will be discussed more in the next chapter.

Materials in BPFL are described either by a symbol or by a property list. The application interpreters had to develop a set of functions that would extract the primary material identifier from a material data value and that would check for the existence of desired properties. There is no specific data type for materials, so the interpreter cannot readily determine whether a symbol or a list was intended to be used as a material specification. This problem is a result of the language definition.

The BPFL lot mechanism was almost fully supported, except for the function that reduces a list of wafer indexes to a list of lot names. The description of a wafer required a structure to save the identifier physically engraved on the wafer and the initial wafer material specification from `allocate-lot`. The access functions for the structure were made available to the application-specific interpreters. The interpreters obtain the structures themselves by giving the logical index for the desired wafer. The application could also query the composition of the `current` lot when it trapped a built-in function

during process execution. The simulation interpreters tracked just one wafer and ignored actions that did not involve the wafer of interest. The work-in-progress interpreter, currently being implemented, has to track all wafers.

For the purposes of the implementation, there are four types of variables: global, local, positional arguments, and keyword arguments. All variables, except global ones, are associated with a particular function call, and thus with a particular `stack-frame`. Global variables are associated with a "null" `stack-frame`. Given this convenient method of identifying variables within a run, the core interpreter stores all variables in a table in the `run` structure. Each variable is indexed by its associated frame, its type, and its name. The names of positional arguments are their numeric indexes. With this schema, the major operations of the core interpreter are straightforward. For example, when a frame is removed from the evaluation stack, all its associated variables are also removed. This operation could also be achieved by storing the variables in the `stack-frame`, but the separate table implementation was chosen because the core interpreter must support interpreters that have to checkpoint part way through interpretation. A checkpoint is much easier if all the variables are in one place.

The core interpreter can be configured for a particular application in three ways. The most common way is to register new built-in function names and the interpreter functions that handle them. The handler function can choose to either receive the BPFL positional arguments as its own, or it can use access functions to interrogate the current `stack-frame`. The direct use of the positional arguments is especially useful when using standard Lisp functions as handlers (e.g., `integerp`, `first` or `list`). If desired, the core interpreter can check the types of the arguments to a built-in function before calling the handler. The core interpreter may also be configured to handle new special forms and constant symbols. These facilities are not currently used by applications, although the core interpreter configures itself with the basic BPFL special forms.

The core interpreter expects BPFL functions to be stored in individual files in a predetermined file-system directory. When the body of a BPFL function is needed, the name of the function was used to form the name of the file. The file is read with a reconfigured version of the standard Lisp reader. The result of reading the file will be a list starting with an appropriate function definition symbol. Since opening and

reading a BPFL function file is sufficiently slow to make a difference with commonly used functions, a function cache was installed. The cache stores the function definition and the last-modified date of the source file. When the function body is requested, the cached function definition is returned if the source file has not been modified. Otherwise, the file is read again.

CHAPTER 4

Experience with the Implementation

This chapter provides some details about the implementation of the interpreters and an evaluation of BPFL.

4.1. Writing the Interpreter

The interpreter system to date contains about 3000 lines of Common Lisp source code for the core interpreter. The SIMPL interpreter contains about 800 additional lines of code, and the SUPREM interpreter contains about 1400 additional lines of code [10].

The Common Lisp implementation is Franz Inc. version 2.0.10 and runs on a DEC μVAX II running the Ultrix operating system and the X10 window system [11] [12] [13]. The SUPREM interpreter was implemented in the Kyoto Common Lisp [14].

In addition to the ease of manipulating the BPFL functions and data, Lisp provides a flexible prototyping environment for developing interpreters. To develop the interpreter, the Lisp-stack printing facility provided sufficient information to find errors in the interpreter, with occasional recourse to the function trace package. A stronger link between the stack printer, the trace package, and the browser would have made viewing circular data structures easier. The Lisp package mechanism was an impediment during development of application interpreters. The core interpreter and application-specific code were in separate packages, which made it difficult to debug both the core and application interpreters at the same time.

The Common Lisp standard unfortunately does not specify a way to trap Lisp errors. The Franz implementation did provide a function for trapping errors, but there was no way to determine the error. Moreover, the code can not be ported to other Common Lisp implementations. To provide error messages that related to BPFL code instead of the depths of the interpreter main loop, it was necessary to check the types of arguments to Lisp functions called from BPFL code. The core interpreter used the Lisp table-driven reader to convert text files containing BPFL functions to internal data structures. If an incorrect function was read, the interpreter would halt, with little indication of where the syntax error was. A hand-

made, robust reader that checked the syntax of a BPFL function would have saved much run-time frustration and would have been worth the development effort.

An object-oriented programming system would have been useful when implementing the BPFL data types for units and intervals. Each standard Lisp function that was extended to accommodate units and intervals had to be implemented by hand in the core interpreter. If the object-oriented concept of specialization had been available, this part of the implementation would have been considerably easier. However, this was a small part of the interpreter and an object system was considered not worth the effort.

The loop in Figure 4.1 executes about two times per second in the core interpreter. The equivalent Common Lisp code (replacing `progn` by `tagbody`) executes about 100 times per second when interpreted. The timing loop creates 10 frames per iteration, so the core interpreter can execute about 20 frames per second. If the core interpreter were used to control manufacturing equipment, this would not be a problem, as little BPFL code is executed between waits on equipment. For the current application of processing an entire process description for conversion to a simulator format, it is only slightly annoying. Although the conversion takes a few minutes, the simulation itself takes much longer. For applications that do not check-point the state of the interpreter, the core interpreter could be rewritten to use the Lisp stack instead of creating an internal frame stack.

4.2. Problems with BPFL

Writing the core interpreter uncovered several problems with BPFL. The biggest contribution to the complexity of the core interpreter was the lack of macros. Instead of defining some of the special forms as macros that expanded to simpler forms (as Lisp does with `cond`, `case`, and loop constructs), the interpreter had to implement each one with a separate dispatch function.

```
(defflow timing (&aux (a 0))
  (progn top
         (setf a (+ a 1))
         (if (< a 100)
             (go top))))
```

Figure 4.1 – Loop used to compare interpreter speeds.

The behavior of the `object` data type was not clearly defined in the BPFL specification. Objects are simple enough to represent within the interpreter as structures storing the object name and a pointer to the contents. However, there is no mechanism for loading the methods defined on the object classes. In particular, methods on objects were never implemented as part of the function-call mechanism. The root cause of this is that the BPFL specification assumed that an implementation would import an object system from the supporting environment. The prototype interpreters did not have such an object system available.

The type system for BPFL was crippled in comparison to Lisp. The predefined types were sufficient to express the data needed to describe processes. However, without being able to specify types, as is needed for type coercion, it was impossible to get around glitches such as a division creating a ratio type when an integer was needed. The ratio data type was probably unnecessary since most integer division desires an integer result instead of a capturing the division as a ratio.

BPFL specifies that no two data structures share any components. The result is that the data values reachable from one variable are disjoint from those reachable from another variable. The lack of sharing between data structures changed the meaning of the `setf` special form. In Lisp, `setf` need only recognize the accessor function and need not analyze the arguments to the function. In BPFL, the accessor and its arguments must be deeply understood to determine the exact data value that is being changed. For example,

```
(setf (first (second a)) 3)
```

requires knowledge about the function `second` in BPFL, but not in Lisp. The core interpreter did not support `setf` for any forms other than variables. The BPFL specification should allow sharing of substructures because it is easier to save shared data than it is to implement the current `setf` semantics.

BPFL poorly specified the meaning of the functions that were extended to accept interval arguments. In particular, the comparison functions took on the Lisp definition of making sequential pair-wise comparisons when more than two arguments were given. Thus, the form

```
(= 1 #i(0 3) 2)
```

would evaluate to `t` because `(= 1 #i(0 3))` is true and `(= #i(0 3) 2)` is true. Clearly the first and last arguments are not equal. The core interpreter implementation forced the comparison of all

arguments to ensure that interval values did not upset the scalar comparisons.

The lot model in BPFL contained some ambiguities. Foremost was the meaning of lots that contained no wafers, either because of attrition or because they were defined that way. Since a run is a process applied to a lot of wafers, it is not clear whether operations on an empty lot are executed. The implementation did not signal an error and simply set the `current` lot to `nil`.

Another major problem with BPFL was that the primitive functions at the flow level could not be easily specialized to accommodate information for similar models. Both the SIMPL and SUPREM interpreters had to add arbitrary keyword arguments to provide local data. While this method worked, clearly it would break down in a more demanding process development environment. BPFL attempts to provide a common frame of reference for all process views. However, it does better at combining views with little in common than it does with views that have small differences.

CHAPTER 5

Future Work

This chapter discusses some possible extensions to the BPFL system. Some changes apply to the interpreters for the current version of BPFL, and some changes are to the language itself.

5.1. Changes to the Interpreter

Currently, the core interpreter reads BPFL function definitions from individual files in a predetermined directory. The intent of this arrangement was that the directory would form a library of functions. However, it is likely that users, especially in a development environment, would not want to do all of their work out of a central library. Thus, the interpreter should be modified to look in user-specified locations for BPFL functions.

Another extension would be to store the functions in a database management system (DBMS). Such an arrangement would provide more flexibility than file based storage and would inherit the management tools of the DBMS. A database would allow browsing previously defined functions based on their properties and finding one that already performs a desired task.

The next major interpreter for BPFL will be the Work-In-Progress (WIP) interpreter, which will operate like a commercial process supervision system. The WIP interpreter will "execute" a process description over a period of days or weeks. To survive possible system crashes the interpreter must be able to save its run-time state for later restoration. For this purpose, the variables and stack of frames in the core interpreter can be written out in a straightforward manner. To continue execution of a BPFL run from where it was suspended, all that is necessary is to read in the frame stack and enter the main interpreter loop, which will then begin dispatching as usual.

Of course, future work could also implement the currently unimplemented features, such as BPFL objects, methods, and the `setf` special form. The WIP system will also require that the "environment" functions, such as `allocate` and `wait-for` be implemented. In some cases BPFL itself should be changed to aid some features.

5.2. Changes to the Language

A great addition to the language would be a macro facility. As mentioned in the previous chapter, many of the dispatch functions could be removed if constructs such as `case` and `while` were implemented in terms of the existing `if` and `go` facilities. Saving the state of the WIP interpreter would not be greatly encumbered by saving expanded macros, although informing the user where errors had occurred would be more difficult.

Most of the generic equipment level primitives take a `:sequence` argument, which is a list that represents a simple program. Using the back-quote (`) list construction syntax of Lisp would simplify parameterizing such lists. An alternative would be to allow the execution of the recipe steps in the appropriate application-specific interpreters. This would make parameterization and variation of recipes easier and would fit into the current built-in function approach to interfacing to the application interpreters.

Currently, the `:implemented-by` facility of BPFL is a major deviation from the basic Lisp semantics and is only used with flow level and generic equipment level primitives. The interpreter could be simplified if users were required to use the `interpreter` predicate to determine when a primitive function was to be called and when the BPFL implementation was to be called. If macros were available, there would be little change in the syntax. For example, the form

```
(grow :material 'silicon-oxide :implemented-by 'oxide-xA375)
```

would expand into something like

```
(if (interpreter 'flow-level)
    (grow :material 'silicon-oxide)
    (oxide-xA375 :material 'silicon-oxide))
```

Thus, the `grow` function would only be called in interpreters with the `flow-level` feature set, in which case `grow` would be a built-in function.

A certain amount of confusion could be avoided if the semantics of BPFL were closer to those of Lisp. Some special forms, such as `progn` and `return` have been changed in subtle ways, and the lack of common components between data structures is counter-intuitive. The one part of Lisp that would be difficult to implement in the WIP interpreter would be the lexical closures. If these were excluded, and if macros and the basic Lisp special forms were brought in, then the language would be much more

convenient and consistent.

Another possible extension to BPFL would be to make the parallel processing of disjoint sublots more explicit. Many machines process wafers individually or in small batches. Capturing this information in the specification would make a WIP interpreter or a factory simulator much more useful.

References

[1] The COMETS system.
Consillium Inc., Mountain View, CA

[2] The PROMIS system.
Promis Systems Corp., Santa Clara, CA

[3] 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

[4] K. Lee
SIMPL-2 (Simulated Profiles from Layout – Version 2)
Memorandum No. UCB/ERL M85/55. July 3, 1985.

[5] *TMA SUPREM-3*
Technology Modeling Associates, Inc.; Palo Alto, CA

[6] Christopher B. Williams, Lawrence A. Rowe.
The Berkeley Process-Flow Language: Reference Document.
Memorandum No. UCB/ERL M87/73. Oct 15, 1987.

[7] Guy L. Steele Jr.
Common Lisp: The Language.
1984, Digital Press, Burlington, MA.

[8] Harold L. Ossher, Brian K. Reid.
Fable: a programming-language solution to IC process automation problems.
Proc. of the SIGPLAN 83 Symp. on Programming Language Issues in Software Systems.
ACM SIGPLAN Notices, Vol. 18, No. 6 (June 1983), pp. 137-148.

[9] Jeff Sedayao.
Personal communication.

[10] David C. Parker
The BPFL to SUPREM Interpreter, Rev 0.1
internal memorandum, Apr. 7, 1988.

[11] *Extended Commmon Lisp User Guide, Release 2.0*
April 1987.
Franz, Inc.; Alameda, CA

[12] The Ultrix operating system.
Digital Equipment Co.
Maynard, MA

[13] R.W. Scheifler, J. Gettys,
"The X Window System,"
ACM Trans. on Graphics, Vol. 5, No. 2 (Apr 1986)

[14] Kyoto Common Lisp
Special Interest Group in LISP
Research Institute for Mathematical Sciences
Kyoto University; Kyoto, 606, JAPAN.