Chapter 4 — Network Queueing Model and Simulator

In this Chapter we present a queueing network model of semiconductor wafer fabrication. We formally define simple and weighted priority rule dispatching. We describe FabSim, a discrete event IC fab simulator, written in C, that allows the simulation of several release strategies and both simple and optimal multi-rule dispatching. The simulator is designed for use in research of shop-floor scheduling policies, but can also be applied to study problems such as fab layout design, capacity analysis, production forecasts, clearing of accumulated inventory, and fab start-up strategy analysis. We number several possible extensions to FabSim that, if incorporated, would improve the package's usability, efficiency, and modeling capability.

4.1 — The Wafer Fabrication Model

In this section we describe a wafer fab model that is implemented as a discrete event simulation engine.

Let fab F be defined as a set of M work stations, each having m_i parallel unreliable machines, $i = 1, ..., M$. Machine i of work station j has time between failures and time to repair that are distributed according to probability distribution functions $G_{u+d}(i,j)$ and $G_d(i,j)$, respectively. Each work station has associated with it a buffer of infinite capacity, from which lots of wafers are selected for processing at that station. Machines can be in one of three states — *idle, down,* or *busy.* When a machine is in working condition it is either *idle* or *busy.* Machine i can process at most c_i lots simultaneously if it is in working condition and cannot process if it is *down.* Equipment failure in non-preemptive, *i.e.* equipment will complete processing of a lot before it is allowed to go down. At time $t = 0$ all machines are assumed *idle* and lot inventory in the fab is zero.

The fab produces p products. Each product p_i, $i = 1, ..., p$ has associated with it a *process recipe* or simply a *recipe.* A recipe defines a sequence of work stations

through which a lot must flow during its fabrication process. The recipe defines, for each work station, the duration of processing, inter-station travel time probability distribution, and the lot rework probability. The recipe also defines the lot start rate and the proportion of high-priority lots of that product to be started. A lot release mechanism triggers lot starts. A lot, upon arriving at a work station, is placed in a buffer of lots waiting for processing at that station. If there are one or more lots waiting in the queue, the system must decide which lot or lots to process next when a machine becomes available. At the end of processing at a given station, the lot is placed in travel status towards its next processing station. The time it remains in travel status is a random variable specified by the recipe.

4.2 – Simple and Weighted Priority Dispatching

In Chapter 1 we define a shop-floor scheduling problem for semiconductor manufacturing. The approach taken in our research towards shop-floor scheduling is two-fold – lot dispatching schemes and lot release strategies. FabSim is the realization of the many strategies that we have pursued. In the dispatching arena we have investigated both simple local rules and more complex rules involving heuristic search. To understand what is available in the package we formalize, in this Section, the notion of dispatching as well as the concept of optimal multi-rule dispatching.

As defined in Chapter 3, *dispatching* is the process by which one or more queued lots are selected for processing at an idle machine. A common way to make this selection is with the use of a *priority dispatching rule.* Denote the set of $n \geq 0$ lots queued at station s by J_s. A priority dispatching rule r is defined uniquely by a priority function P_r: $J_s \rightarrow \mathbf{R}$ that assigns real values to elements of J_s. Rule r selects the lot with the smallest priority function value, *i.e.*, the lot with the highest priority. Denote $I_r(s)$ to be the index of the lot selected, *i.e.*,

$$I_r(s) = \text{argmin}\left\{ P_r(j) : j \in J_s \right\}. \tag{4.1}$$

A *normalized* priority function is a priority function $\bar{P}_r$: $J_s \rightarrow [0, 1]$ that assigns real values between 0 and 1 to elements of J_s.

If a batch operation is called for and more than one lot is to be selected, rule r is repeatedly applied to a feasible sub-set of the remaining queued lots until the required number of lots are chosen.

Table 4.1 illustrates the above notation with some classical priority dispatching rules.

Rule	$P_r(j)$	Description	I_r^s
FIFO	$A_s(j)$	Arrival time of job j at queue of station s.	Lot that has been most time in queue of station s.
RANDOM	$U(0,1)$	Uniformly distributed number $\in$ (0,1).	Lot at queue of station s selected at random.
SIPT	$p_s(j)$	Processing time of job j at station s.	Lot at queue of station s with shortest processing time at that station.
SRPT	$r_s(j)$	$p_s(j)$ + sum of processing times of job j on all future recipe steps.	Lot at queue of station s with shortest remaining processing time.

Table 4.1 - Priority Function

As was described in Chapter 3, a priority dispatching rule can be defined to be a weighted sum of two or more rules. Let R be the set of N rules to be considered,

$$R = \left\{ r_1, r_2, \ldots, r_N \right\}, \tag{4.2}$$

and let P_r be the priority function associated with rule r. Define a weighted priority rule, denoted by $\hat{r}(\mathbf{w})$, to be such that its priority function is given by

$$\hat{P}(j) = \sum_{i=1}^{N} w_i P_i(j) \tag{4.3}$$

where $\mathbf{w} \in \mathbf{R}^N$. Every input vector $\mathbf{w}$ defines a priority dispatching rule $\hat{r}(\mathbf{w})$. An obvious question to ask is — Which vector $\mathbf{w}^*$ defines the *best* dispatching rule $\hat{r}(\mathbf{w}^*)$?

To answer this question we require a model of the system under consideration. Let S be a simulation model of fab F with controllable input vector $\mathbf{w} \in \mathbf{R}^N$, uncontrollable input vector $\mathbf{z} \in \mathbf{R}^p$ and a measured output

$$\xi = g(\mathbf{w}, \mathbf{z}), \tag{4.4}$$

such as mean lot delay. It is important to note that ξ is only an observed value and may not correspond to the true system output y.

By repeated measurements of the output ξ, an estimate $\hat{y}$ of the true system response can be computed according to

$$\hat{y} = \frac{1}{M} \sum_{i=1}^{M} \xi_i = y + \epsilon \tag{4.5}$$

where M is the number of output measurements, ξ_i is the i-th observed output, and ϵ is the estimation error, assumed to have normal distribution with mean zero and variance $\sigma^2 = \mathrm{Var}(\hat{y})$.

If we let

$$y = h(\mathbf{w}) \tag{4.6}$$

we can rewrite the estimate $\hat{y}$ as

$$\hat{y} = h(\mathbf{w}) + \epsilon \tag{4.7}$$

and therefore

$$E(\hat{y}) = y \tag{4.8}$$

and

$$Var(\hat{y}) = Var(\epsilon). \tag{4.9}$$

Our objective is to optimize the expected value of output estimate of the true system, $E(\hat{y})$. To do so, we define an optimization problem.

$$\text{opt } y = h(w) \tag{4.10}$$

$$\text{subject to: } w \in R^N \tag{4.11}$$

As discussed in Chapter 3, several techniques have been proposed for solving optimization problem (4.10-4.11). We select to implement Hooke and Jeeves pattern search due to its prior successful application in the context of job shop scheduling, *e.g.* Hershauer and Ebert [Her75a], Harrison and O'Grady [Har85a], and Bunnag and Smith [Bun85a].

4.3.1 — FabSim - A Discrete Event Fab Simulator

The fab model described in the previous section has been implemented as a discrete event simulator — FabSim. (See Law and Kelton [Law82a] for a discussion of computer simulation.) The code is written in C. Presently, versions of the code run on the following machines —

- VAX 11/750 and 8800 under BSD 4.3 UNIX, ULTRIX and VMS
- Micro-VAX II under ULTRIX and VMS
- IBM 3090 running VM/SP CMS (with a Waterloo C compiler)
- Cray X-MP/14 under UNICOS

In this Section data structures and logic of the code are briefly described. A list of available dispatching rules and release strategies is provided.

The program allows for two classes of dispatching schemes — with or without rule mix optimization. If optimization is required a variant of Hooke and Jeeves pattern search is performed with calls to the simulation engine. If no optimization is called for the program makes one call to the simulation engine. We begin by describing the simulator and then discuss our implementation of pattern search.

The simulation engine is designed to work as a list processor. All entities are represented as linked lists. The following lists are manipulated by the simulator.

- *product_recipe*— A linked list whose elements contain information about a single step of a product recipe. This includes step duration, step number, work station number, rework probability, yield, mean travel time to next station, and a pointer to its corresponding work station element in the list of work stations.
- *lot*— A linked list whose elements contain information about a single lot in the fab. This includes product type, lot number, hot-lot indicator, time into fab, time into present queue, time left in current equipment, priority in queue, cumulative queue time, cumulative yield, lot due date, and a pointer to its current position in the product recipe list.
- *work_station*— A linked list whose elements contain information about a single work station in the fab. Its elements contain station number, current queue size, equipment load size, cumulative queue size, pointers to the first and last elements of its queue, a pointer to the first element of the list of equipment of the work station, and a pointer to the first and last elements of the list of lots currently traveling from that station to the some other station in the fab.
- *queue*— A linked list representing a queue in a work station. Its elements contain a pointer to a lot element in the list of lots in the fab.

- *equipment*— A linked list of equipment in a given work station. Its elements contain equipment status indicator, time until equipment is to go down, time until equipment is to come back up, time until processing of current lot(s) is to terminate, cumulative time busy, cumulative time idle, cumulative time down, mean down time, mean up time, a pointer to the first element of the list of lots currently being processed by it, and a pointer to the element in the work station list that corresponds to its work station.
- *in_process*— A linked list of lots being currently processed by some equipment. Its elements contain a pointer to a lot cell in the list of lots in the fab.
- *travel*— A linked list, pointed to by a work station, of lots currently traveling from that station to some other station in the fab. Its elements contain a pointer to a lot cell in the list of lots in the fab and the lot's arrival time at the next station.

The simulation is event driven, *i.e.* time in the simulation is not incremented in units, but rather scheduled events are kept sorted in a heap data structure [Sta80a] and time is incremented to the epoch of the next scheduled event. The simulation engine is described below in pseudo-code.

```
procedure simulate()
begin
        set_up()
        update_times()

        while (simulation not done)
        begin
                lot_start_check()
                load_travel_check()
                load_queue_check()
                equip_status_check()
                load_equip_check()
                update_times()
        end
        output_reports()
end.
```

The set_up() routine sets up the program for running the simulation. In this routine the lists are initialized, problem data read, and initial simulation events are scheduled. Input includes a description of the fab and its products, and a set of simulation parameters that includes random number generator seeds, simulation horizon, release strategy, dispatching rule, residence time distribution, travel time distribution, reports to be generated, and due date setting rule. The description of the fab is given by a set of work stations, each with work station number, number of equipment in work station, equipment load size, an indication of whether the station is a hub station, and a list of equipment in each work station, with estimates of mean time to repair (MTTR) and mean time between failures (MTBF) for each piece of equipment. FabSim assumes that both MTTR and MTBF are random variables with exponential distributions. The description of the products is defined by the number of products, and for each product, the number of steps in its recipe, its wafer start rate, its profit value, the proportion of lots of that product that are hot, a due date setting parameter, and a list of steps that constitute the product's recipe, with each step having a step number, a work station where the step is to be performed, the mean step duration in minutes, the rework probability, the step yield, and the mean travel time from the current station to the next station in the recipe. Travel time can be either deterministic or random with exponential distribution.

With the present version of FabSim the following dispatching rules are available —

FIFO First-In-First-Out — Select the lot that first arrived at the queue.

SIPT Shortest Imminent Processing Time — Select the lot with the shortest processing time at the current station.

SIPT/T Truncated Shortest Imminent Processing Time — If no lot has been T or more time units in the queue, select the lot with the shortest processing

time at the current station. Otherwise, select the lot that has been in the queue the longest, *i.e.* use FIFO.

LIPT Longest Imminent Processing Time — Select the lot with the longest processing time at the current station.

SRPT Shortest Remaining Processing Time — Select the lot with the shortest processing time for the rest of its operations in the fab.

LRPT Longest Remaining Processing Time — Select the lot with the longest processing time for the rest of its operations in the fab.

SIPT/V Smallest Ratio of Imminent Processing Time to Product Value — Select the lot with the smallest ratio of processing time to product value at the current station.

SRPT/V Smallest Ratio of Remaining Processing Time to Product Value — Select the lot with the smallest ratio of processing time for the rest of its operations in the fab to product value.

LNOP Least Number of Remaining Operations — Select the lot with the least number of remaining operations for the rest of its stay in the fab.

NINQ Smallest Number In Next Queue — Select the lot whose next station has the queue with the smallest number of waiting lots.

NINQ/M Smallest Ratio of Number In Next Queue to Number of Equipment — Select the lot whose next station has the queue with the smallest ratio number of waiting lots to number of equipment.

RAND Random — Select the next lot at random.

NEDNS Smallest Number of Equipment Down at Next Station — Select the lot whose next station has the smallest number of equipment down .

SCNQ Smallest Expected Queue Clear Time — Select the lot whose next station has the queue with smallest expected time to clear. This rule takes into

account the work remaining in queue and the number of down machines at the station.

SERT Shortest Expected Remaining Time in System — Select the lot with the shortest expected remaining time in the fab. Expected time remaining in system is computed by adding the remaining processing time with the expected remaining time in queue. Expected remaining time in queues is the sum of the expected time to clear queue for each queue remaining in the job's recipe. This is approximated by dividing the queue work content by the virtual number of machines at the queue's station (= number of machines x average machine availability).

DSERT Shortest Discounted Expected Remaining Time in System — Select the lot with the shortest expected discounted remaining time in the fab. This is similar to SERT with the difference that expected queueing time is discounted.

DQCT Discounted Queue Clear Time — Select the lot with the discounted expected remaining queue clear time. This is similar to DSERT except that job processing times are disregarded.

DD Earliest Due Date — Select the lot with the earliest due date. Due dates are currently set using the TWK rule, which sets the due-date of a job to $k \times p_j$ units of time after it enters the shop, where $k \geq 1$, and p_j is the job's total processing time. Baker [Bak84a] describes TWK and other due-date setting rules.

MULTI Multiple rule — Given a set of simple rules and a vector of weights, MULTI combines the simple rules according to the given weights and selects the lot given the highest priority be the weighted priority function.

SA Starvation Avoidance rule — This rule combines, via the MULTI rule,

SRPT with a rule that prioritizes lots according to the ratio of duration of next bottleneck station visit to time until next bottleneck station visit. The weights are adjusted dynamically so as to give more weight to SRPT when the bottleneck is no danger of running out of work or to the ratio rule described above when there is imminent danger of starving the bottleneck. This rule is described in detail in Chapter 5.

OPTMIX Optimal multi rule — Given a set of simple rules and a vector of weights, OPTMIX combines the simple rules and uses Pattern Search to find a vector of weights that produces a good dispatching rule.

The update_times() routine finds the next event in the heap of scheduled events and updates the simulation clock and count-down times in the list structures. Count-down times include time until equipment is scheduled to go down, time until down equipment is scheduled to come up, time until a process is to finish, and time until the next lot is started.

The *while loop* is run while there are still lots in the fab that were started before the simulation horizon H. In the loop the program first checks, in lot_start_check(), if a new lot is to be started. This can be done in several ways. The program allows for the following release strategies —

POIS Poisson — Interarrival times for each product are distributed exponentially.

UNIF Uniform — Interarrival times for each product are fixed.

Fixed-WIP Fixed WIP — The number of lots in the fab is kept constant. When a lot of a given product departs, another of the same product is started.

WR Workload Regulation — This rule is due to Wein [Wei86a]. It releases a lot into the fab when the remaining work at the bottleneck for all lots in

the fab falls below a prescribed value.

SA Starvation Avoidance – This rule attempts to avoid starvation of any bottleneck machine. It is described in detail in Chapter 5.

If a lot is to be started, a new lot is defined, loaded into the queue of the work station of its first recipe step and (if the release policy is either POIS or UNIF) a new lot start for that product is scheduled and inserted in the event heap. In load_queue_check() equipment lists are scanned and if a lot is finished processing and no rework is required it is placed in the travel buffer of lots currently traveling out of this work station. An end travel event is scheduled and placed in the event heap. If rework is called for the lot is replaced at the head of the queue of lots waiting for processing at that station. In load_travel_check() travel buffers are scanned and if a lot is done traveling, it is removed from the buffer and is placed in the queue of the next work station in its recipe or is removed from the fab if it has just completed its last step. If the lot is removed from the fab, statistics are collected. Job statistics are collected for jobs started between times $\epsilon_1 H$ and $\epsilon_2 H$, where $0 \leq \epsilon_1 < \epsilon_2 \leq 1$ and H is the simulation horizon. Station statistics are collected beginning at time $\epsilon_1 H$. The intent here is to avoid any bias (due to light traffic) before collection of lot statistics begins. In equip_status_check() all work stations are scanned for an equipment that is *idle* and scheduled to go *down* or one that is *down* and scheduled to come up, thus becoming *idle.* If found, equipment status is changed and new events scheduled and inserted in the event heap. In load_equip_check() work stations are checked for an *idle* piece of equipment and for waiting lots at the station buffer. If such a situation is identified, lots in the queue are ordered according to their priorities and one or more lots are selected for loading into the equipment. Lots in the same batch are constrained to be at the same recipe step as the first loaded lot (*i.e.* the lot with the highest priority). No more lots than the maximum load size of the equipment can be loaded at one time. The selected lots are

removed from the work station queue list and put into the equipment in_process list. An end processing event is scheduled and placed in the event heap. The iteration ends with an update_times() where the next event is retrieved from the event heap and the simulator clock and count-down times are updated.

Routine output_report() generates a report of the simulation run.

4.3.2 – Pattern Search

FabSim allows the choice of optimal multi-rule priority dispatching, or OPTMIX dispatching. To allow OPTMIX, FabSim makes use of Hooke and Jeeves [Hoo61a] pattern search. A detailed description of the Hooke and Jeeves pattern search algorithm implemented in FabSim is given in this Section.

As input to the algorithm one must provide —

- a simulation model S of a fab F, with simulation horizon H.
- a set of dispatching rules R, with normalized priority functions,
- an initial weight vector $\mathbf{w}_0 \in \mathbf{R}^N$,
- an initial step size δ_0,
- a minimum step size $\underline{\delta}$,
- a maximum step size $\bar{\delta}$,
- a shrinking factor ρ, $0 < \rho < 1$.

Pseudo code for the main procedure of the algorithm is given next. This procedure is called *patternsearch*.

---

procedure patternsearch(R, w_0, δ_0, $\underline{\delta}$, $\bar{\delta}$, ρ)

begin
Set **n** = **w** = w_0
Set quit = FALSE
Set $\delta = \delta_0$
Compute h_n = simulate(**n**)
Set $h_w = h_n$
Compute h_n = explore(**n**, h_n)

while quit = FALSE **do**
begin
if $h_n < h_w$
then compute h_n = patternmove(**w**, **n**, t, h_w,h_n,h_t)
else compute h_n = homein(**w**, **n**, t, h_w,h_n,h_t,quit)
Set **n** = 2**n** − t
Compute h_n = simulate(**n**)
Compute h_n = explore(**n**, h_n)
end
return(**w**, h_w)
end.

Pattern search begins with an initial weight vector w_0. Two processes are available in FabSim for generating w_0 — the first generates *n* points at random and select the best as w_0 while the second selects as w_0 the best point out of the points $e_1, e_2, ..., e_N$, where e_i has zeroes on all components except the i-th, which is unit. All point evaluations are carried out by calling procedure *simulate.* This procedure runs the simulation engine described earlier using weighted dispatching rule $\hat{r}$(w).

---

procedure simulate(w)

begin
Run fab simulation using dispatching rule $\hat{r}$(w).
Measure output ξ.
return(ξ)
end.

At each iteration of *patternsearch*, *i.e.* when an improved point w is found, the algorithm (using procedure *explore*) generates several points in the neighborhood of w and returns the best point, n, with respect to the objective function under consideration.

---

procedure explore(n, h_n)

```
begin
        for k = 1 to N do
        begin
                Set n[k] = n[k] + δ
                Compute h_t = simulate(n)

                if h_t ≥ h_n then
                begin
                        Set n[k] = n[k] - 2δ
                        Compute h_t = simulate(n)

                        if h_t ≥ h_n
                                then set n[k] = n[k] + δ
                                else set h_n = h_t
                end
                else set h_n = h_t
        end
        return(h_n)
end.
```

If the weight vector n is superior (in terms of the objective function) to w, a pattern move is carried out. This is implemented in procedure *patternmove*. Otherwise

a step size change action is taken in procedure *homein*.

In the pattern move, weight vector 2n - w is evaluated. If it is superior to n, then w is set to it. Else w is set to n.

---

procedure patternmove(w, n, h_w, h_n)

```
begin
        Set δ = δ0
        while hn < hw do
        begin
                Set t = w
                Set w = n
                Set hw = hn
                Set n = 2n − t
                Compute hn = simulate(n)
                Compute hn = explore(n,hn)
        end
        return(hn)
end.
```

In the step size change procedure, *homein*, the current step size is initially reduced, and exploratory search is carried out around the current weight vector until either an improvement is found or the allowable number of candidate vectors are tested. In the former case w is set to the improving vector, the step size is reset to its initial value δ_0 and control is returned to procedure *patternsearch*. In the latter case, the step size is further reduced (if possible) and another exploratory search is conducted. If no further reduction of the step size is possible, *i.e.* $\delta < \underline{\delta}$, the procedure begins its expand out phase. In this phase, the step size is increased and an exploratory search is carried out. Again, an improvement may be discovered. If this is the case w is set to this improved vector, the step size is reset to its initial value δ_0 and control is returned to procedure *patternsearch*. If no improvement is found the step size is further increased, if possible, and another exploratory search is conducted. If

no further increase is possible, *i.e.* $\delta > \bar{\delta}$, the procedure returns a signal to *patternsearch* to terminate.

---

procedure homein(w,n, h_w, h_n, quit)

begin
    Set $\delta = \rho\delta$

    if $\delta < \underline{\delta}$ **then**
    begin
        Set $\delta = \delta_0$
        Compute h_n = explore(n, h_n)

        while $h_n \geq h_w$ **do**
        begin
            Set $\delta = \delta/\rho$

            if $\delta > \bar{\delta}$ **then**
            begin
                Set quit = TRUE
                break (from **while** loop)
            end
            else compute h_n = explore(n, h_n)
        end

        if quit = FALSE **then**
        begin
            Set $\delta = \delta_0$
            Set $h_w = h_n$
            Set t = w
            Set w = n
        end
    end
    return(h_n)
end.

4.3.2.1. — Normalized Priority Functions

In the input list of our implementation of pattern search we include a set of dispatching rules R, with *normalized* priority functions. The requirement that the priority functions be normalized is practical in nature. Since the step sizes and shrinking fac-

tor are independent of the magnitude of the values of the priority functions, the algorithm frequently stalls in a non-optimal point when the dispatching rules of set R have values with large disparities.

In our implementation we normalize the values by maximum possible value. For example, to compute the normalized SRPT rule we first compute for every station which process step at that given station has the maximum remaining processing time. Then, during dispatching all remaining processing time values are divided by the above maximum value. While this approach is simple, it has been able to improve the performance of pattern search substantially. This normalization, however, is rule dependent and may not be defined for some rules (*e.g.* FIFO). In our implementation we have defined normalized rules for the majority of the available dispatching rules.

4.4 – Extensions

Several extensions are planned for FabSim. These enhancements are designed to improve the simulator's modeling capabilities, efficiency, and usability.

To improve its modeling capability five items are of immediate importance – the modeling of setup and changeover times, the modeling of fab operators as a allocable resource, the ability to have rework loops that involve an extra step, such as layer scrape, and loops with one or *more* work stations, a greater variety of probability distributions, and the capability of restricting processing (for critical layers, for example) to a sub set of a station's machines.

As is the case with most simulators, efficiency is important. Portions of the code should be modified to enhance code efficiency. Research into parallel computer architectures for simulation should be pursued.

A user interface is needed. This interface should allow the user to easily define a simulation environment, design a simulation experiment, control the execution of

the simulation experiment, and manipulate both graphical and alpha-numeric simulation results.

4.5 – Summary

In this Chapter we introduced a queueing model of the dynamics of a semiconductor wafer fab and presented a C programming language implementation of the model. The model presented is a generalization of the classical job shop allowing for unreliable parallel machines, multiple products, deterministic and stochastic (rework) job routes, deterministic or stochastic inter-station travel times, regular- and high-priority (hot) jobs, and two levels of scheduling control – job dispatching and job release.

The C implementation of the model is a discrete-event simulation program called FabSim. Every aspect of the model is implemented in FabSim. Moreover, the program allows numerous dispatching and release control combinations, including optimal multi-rule dispatching with Hooke and Jeeves pattern search for optimization. Detailed statistics are collected and displayed for analysis.

Finally, several improvements to both the model and the simulation program are suggested.