Systems:

- *Continuous systems*
  State changes continuously in time (e.g., in chemical applications)
- *Discrete systems*
  State is observed at fixed regular time points (e.g., periodic review inventory system)
- *Discrete-event systems*
  The system is completely determined by random event times $t_1, t_2, \ldots$ and by the changes in state taking place at these moments (e.g., production line, queueing system)

Time advance:

- Look at regular time points $0, \Delta, 2\Delta, \ldots$ (*synchronous* simulation); in continuous systems it may be necessary to take Δ very small
- Jump from one event to the next and describe the changes in state at these moments (*asynchronous* simulation)

We will concentrate on asynchronuous simulation of discrete-event systems

Terms often used:

- **System**
  Collection of objects interacting through time (e.g. production system)
- **Model**
  Mathematical representation of a system (e.g., queueing or fluid model)
- **Entity**
  An object in a system (e.g., jobs, machines)
- **Attribute**
  Property of an entity (e.g., arrival time of a job)
- **Linked list**
  Collection of *records* chained together

- **Event**
  Change in state of a system
- **Event notice**
  Record describing when event takes place
- **Process**
  Collection of events ordered in time
- **Future-event set**
  Linked list of event notices ordered by time (FES)
- **Timing routine**
  Procedure maintaining FES and advancing simulated time

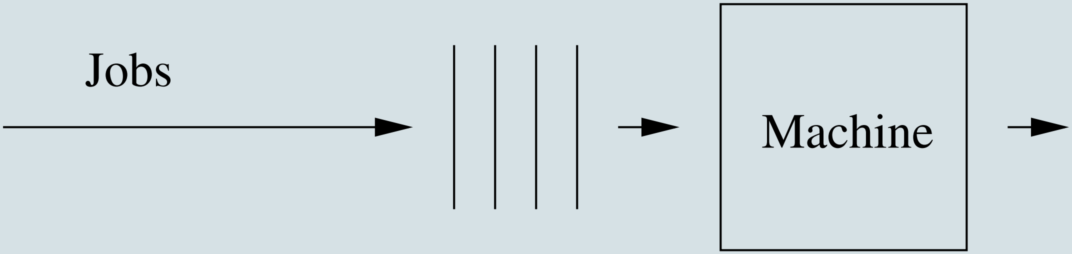
Basic approaches for constructing a discrete-event simulation model:

- *Event-scheduling approach*
  Focuses on events, i.e., the moments in time when state changes occur
- *Process-interaction approach*
  Focuses on processes, i.e., the flow of each entity through the system

In general-purpose languages one mostly uses the event-scheduling approach; simulation languages (e.g., χ) use the process-interaction approach

Event-scheduling approach

Example: Single-stage production system



A single machine processes jobs in order of arrival. The interarrival times and processing times are exponential with parameters λ and μ (with $\lambda < \mu$).

- What is the mean waiting time?
- What is the mean queue length?
- What is the mean length of a busy period?
- How does the performance change if we speed up the machine?

Discrete simulation:

A_n the interarrival time between job n and $n+1$

B_n the processing time of job n

W_n the waiting time of job n

Then (Lindley's equation):

$$W_{n+1} = \max(W_n + B_n - A_n, 0)$$

Initialization

```
n = 0 {job number}
w = 0 {waiting time of job n
       we assume that initially the system is empty}
sum_w = 0 {sum of all waiting times upto job n}
```

Main program

```
while (n < N)
do
    a = interarrival_time
    b = service_time
    w = max(w + b - a, 0)
    sum_w = sum_w + w
    n = n + 1
end
```

Output

```
Mean waiting time = sum_w / N
```

Discrete-event simulation:

Entity	**Attribute**
Job	Arrival time
Machine	Status (idle or busy)

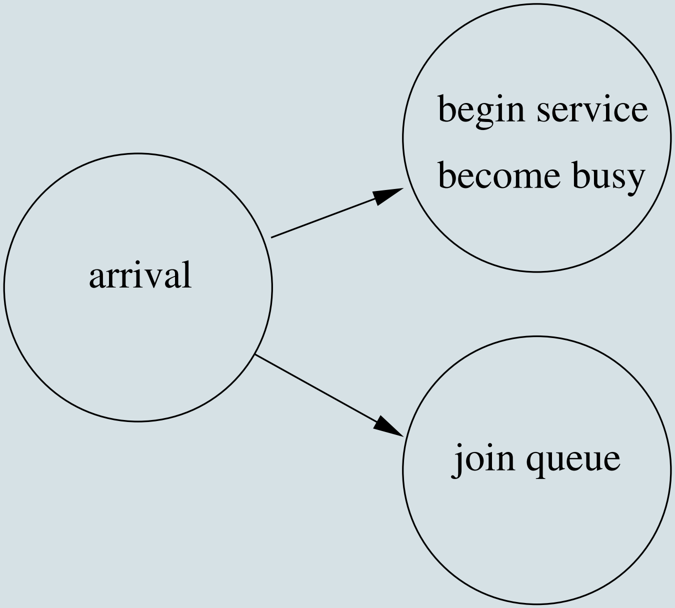
Job is a *temporary* entity
Machine is a *permanent* entity

Elementary events

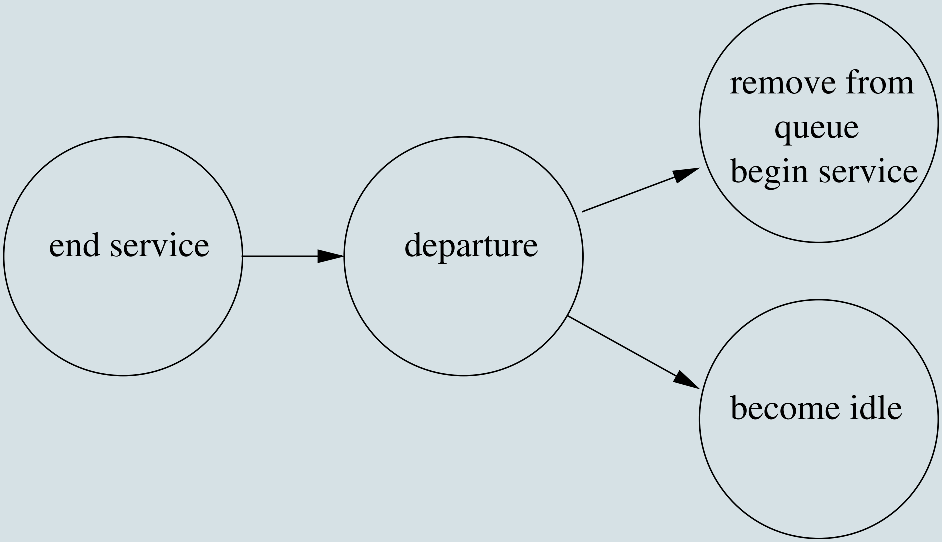
Job:	**Machine:**
arrival	remove from queue
departure	become busy
begin service	become idle
end service	
join queue	

Compound events

Arrival



Departure


end service
departure
remove from queue
begin service
become idle

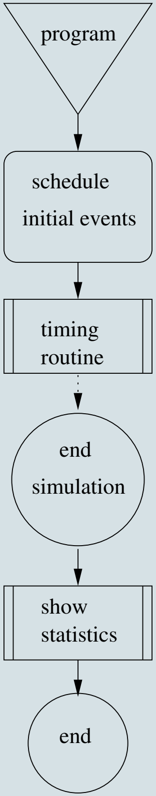
State of the system at time t:

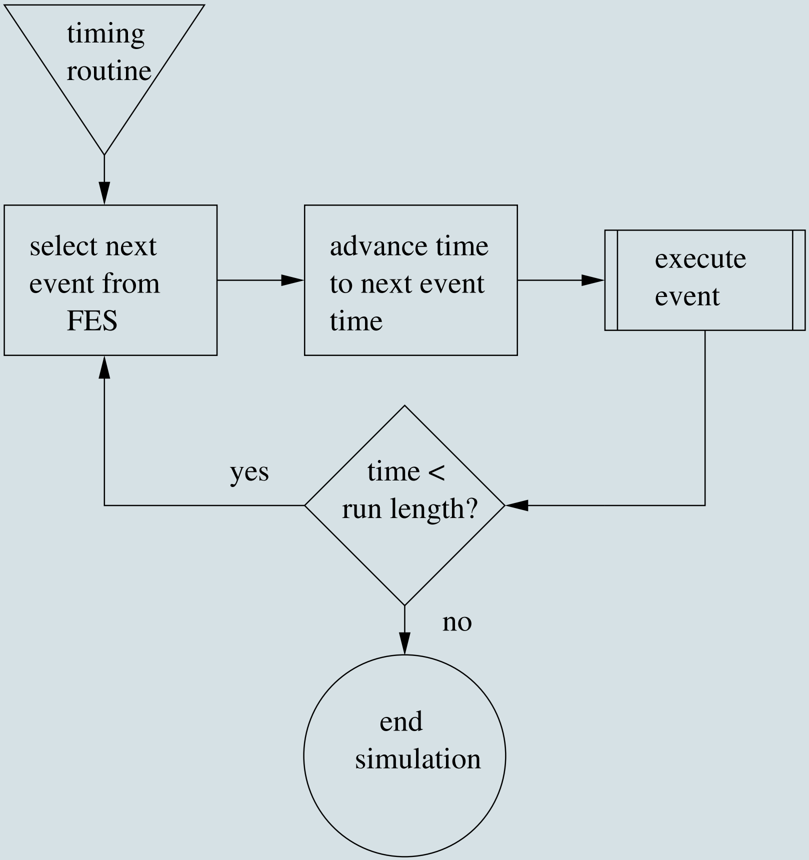
- status of the machine ($i = 0, 1$)
- number of jobs in the queue ($n = 0, 1, 2, \ldots$)
- remaining interarrival time ($a \geq 0$)
- remaining service time ($b \geq 0$)

Then the remaining time until the next event is given by

$$\min(a, b)$$

Prototypical event-scheduling approach:



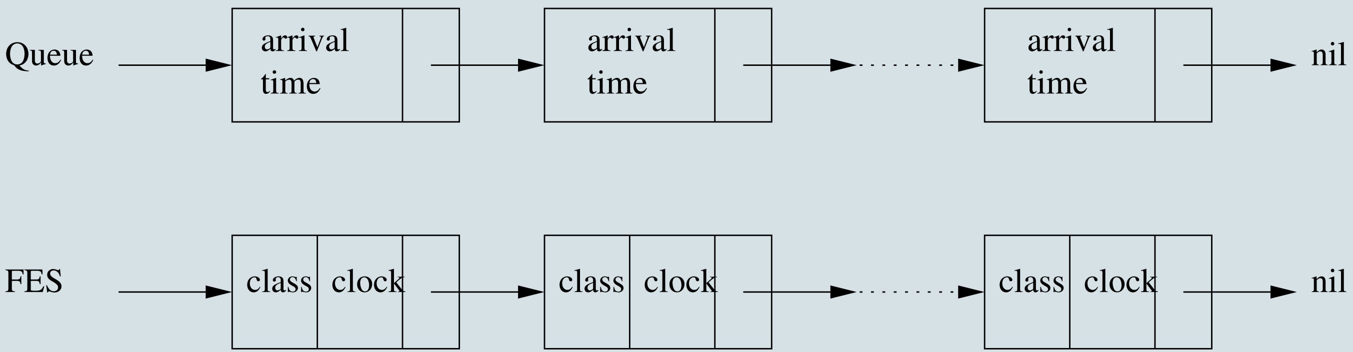

timing
routine
select next
event from
FES
advance time
to next event
time
execute
event
yes
time <
run length?
no
end
simulation

Record Job = (arrival time, ..., successor address)

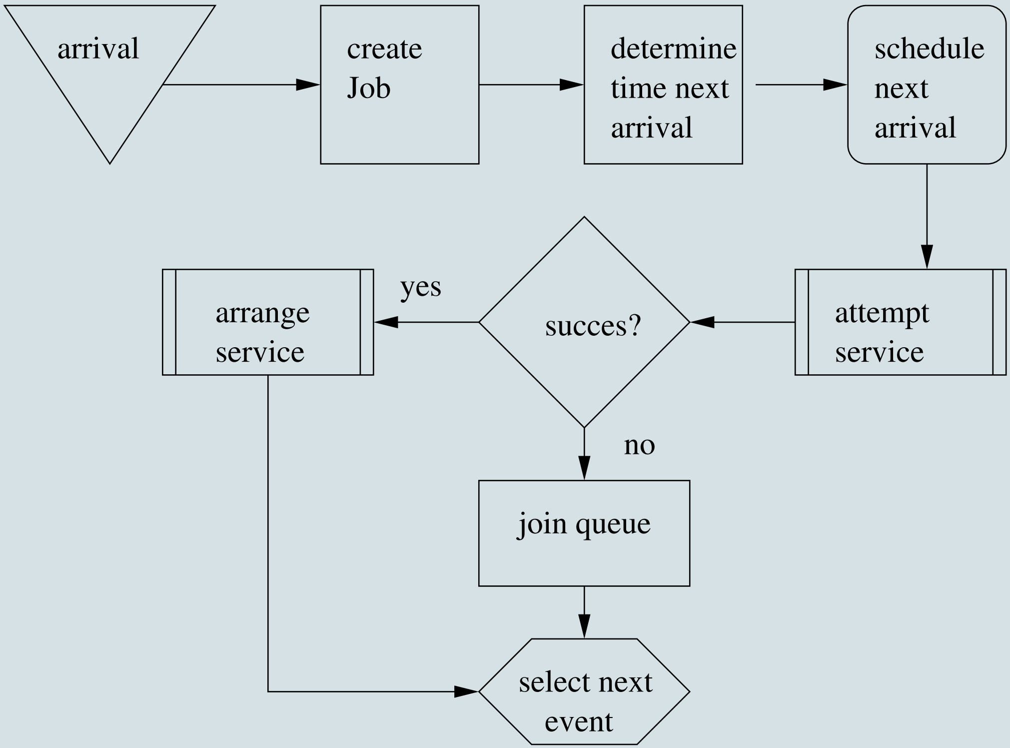
Record Event = (class, clock, ..., successor address)

The queue is a linked list of Job records ordered according to arrival time

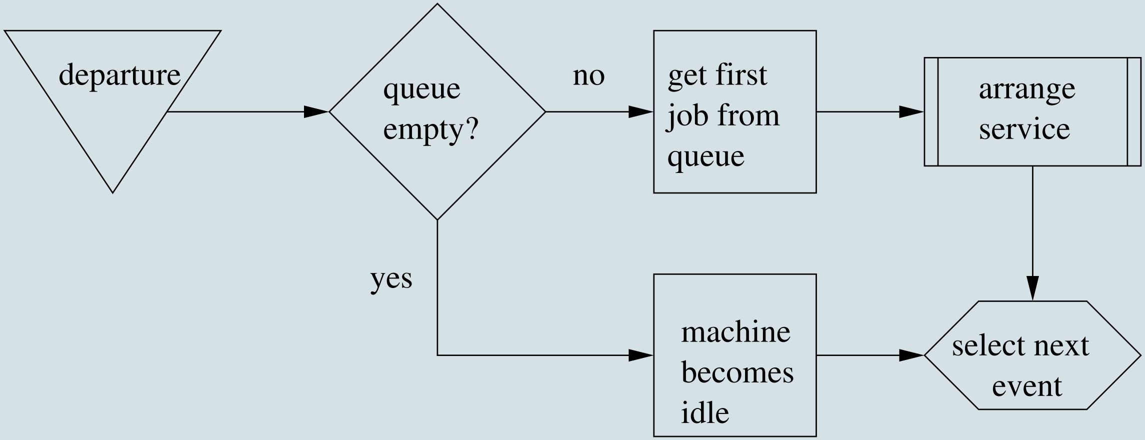
The FES is a linked list of Event records ordered according to clock time



Arrival event:



Departure event:



Initialization

```
t = 0       {current time}
queue = nil {queue is empty}
generate and schedule first arrival
N = 0       {number of jobs processed}
sum_w = 0   {sum of waiting times of processed jobs}
```

Main program

```
while (t < run_length)
do
  determine next_event
  t = event_time
  case next_event of
    arrival_event:
      generate and schedule next arrival
      if machine = busy
      then create and add job to queue
      else
        machine = busy
        N = N + 1
        generate and schedule next departure
```

```
    departure_event:
      if queue not empty
      then
        get first job from queue
        N = N + 1
        sum_w = sum_w + waiting_time
        generate and schedule next departure
      else machine = idle
end
```

Output

```
Mean waiting time = sum_w / N
```

Implementation in C

Definition of records: Events and Jobs

```
typedef struct job {
    double  arrival_time;
    struct job *next_job;
}

        job;

typedef struct event {
    int     class;
    double  clock;
    struct event *next_event;
}

        event;

event   *FES,           /* linked list of events */
        *Used_events;   /* linked list of used event notices */
job     *Queue,         /* linked list of jobs */
        *Used_jobs;     /* linked list of used job records */
```

Operations on the FES: create and destroy

```
event  *create_event()
{
    event  *temp;

    if (Used_events == NIL)
        return (event *) malloc(sizeof(event));
    else {
        temp = Used_events;
        Used_events = Used_events->next_event;
        return temp;
    }
}

void    destroy_event(event * pntr)
{
    pntr->next_event = Used_events;
    Used_events = pntr;
}
```

Operations on the FES: next and add

```
event  *next_event()
{
    event  *pntr;

    if (FES == NIL)
        return NIL;                              /* FES is empty */
    else {
        pntr = FES;
        FES = FES->next_event;
        return pntr;
    }
}
```

```
void    add_event(event * pntr)
{
    event  *link,
           *prev;

    if (FES == NIL) {
        FES = pntr;
        FES->next_event = NIL;
    } else {
        if (pntr->clock <= FES->clock) {
            pntr->next_event = FES;
            FES = pntr;
        } else {
            prev = FES;
            link = FES->next_event;
            while (link != NIL && pntr->clock > link->clock) {
                prev = link;
                link = link->next_event;
            }
            prev->next_event = pntr;
            pntr->next_event = link;
        }
    }
}
```

Initialization

```
void    initialization()
{
    srand48(seed);

    t = 0.0;
    busy = FALSE;
    Queue = NIL;
    Used_jobs = NIL;

    /* initialize FES */
    FES = create_event();
    FES->class = ARRIVAL;
    FES->clock = interarrivaltime();
    FES->next_event = NIL;

    Used_events = NIL;

    N = 0;
    sum_w = 0.0;
}
```

Main program

```
main()
{
    event  *pntr;

    getinput();
    initialization();

    while (t < run_length) {
        pntr = next_event();
        t = pntr->clock;                        /* advance time */
        switch (pntr->class) {
        case ARRIVAL:
            arrival_event();
            break;
        case DEPARTURE:
            departure_event();
            break;
        case NIL:
            printf("FES is empty\n");
            exit(1);
            break;
        }
        destroy_event(pntr);
    }

    output();
}
```

Compound event Arrival

```
void    arrival_event()
{
    event  *pntr_event;
    job    *pntr_job;

    pntr_event = create_event();                  /* schedule next arrival */
    pntr_event->class = ARRIVAL;
    pntr_event->clock = t + interarrivaltime();
    add_event(pntr_event);

    if (busy) {
        pntr_job = create_job();
        pntr_job->arrival_time = t;
        add_job(pntr_job);
        if (Queue == NIL)
            printf("queue is nil\n");
    } else {
        busy = TRUE;
        N ++;
        pntr_event = create_event();
        pntr_event->class = DEPARTURE;
        pntr_event->clock = t + servicetime();
        add_event(pntr_event);
    }
}
```

Compound event Departure

```
void    departure_event()
{
    double  waiting_time;
    event  *pntr_event;
    job    *pntr_job;

    if (Queue != NIL) {
        pntr_job = next_job();
        N ++;
        waiting_time = t - pntr_job->arrival_time;
        sum_w += waiting_time;
        destroy_job(pntr_job);
        pntr_event = create_event();            /* schedule next departure */
        pntr_event->class = DEPARTURE;
        pntr_event->clock = t + servicetime();
        add_event(pntr_event);
    } else                                      /* Queue is empty */
        busy = FALSE;
}
```

Proces-Interaction approach

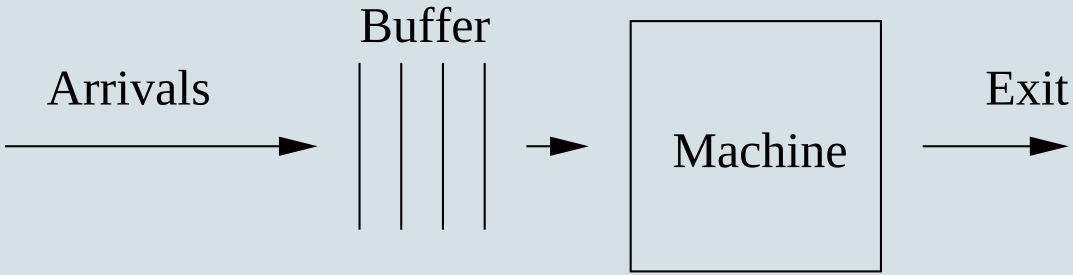
This approach focusses on describing *processes*;
In the event-scheduling approach one regards a simulation as executing a sequence of events ordered in time; but *no time elapses* within an event.

The process-interaction approach provides a process for *each entity* in the system; and *time elapses* during a process.

In production systems we have processes for:

- Arrivals
- Buffers
- Machines
- Exit

Example: Single-stage production system

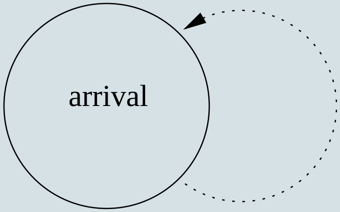


A single machine processes jobs in order of arrival. The interarrival times and processing times are exponential with parameters λ and μ (with $\lambda < \mu$).

- What is the mean waiting time?
- What is the mean queue length?
- What is the mean length of a busy period?
- How does the performance change if we speed up the machine?

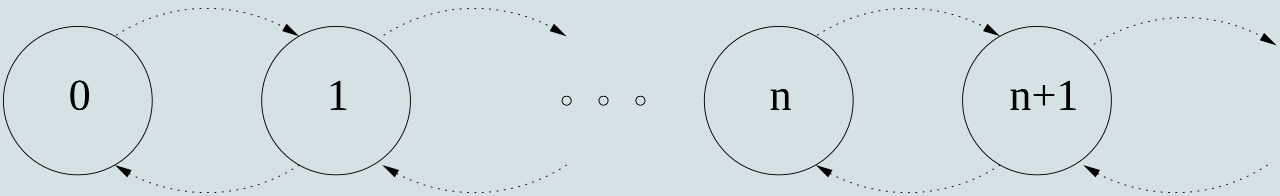
Arrival process

Generate arrival after random (exponential) time units



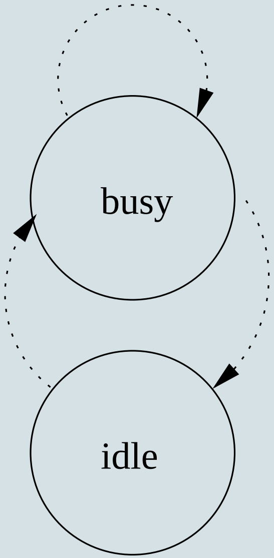
Buffer process

Add job to buffer and remove job from buffer (if there is any)



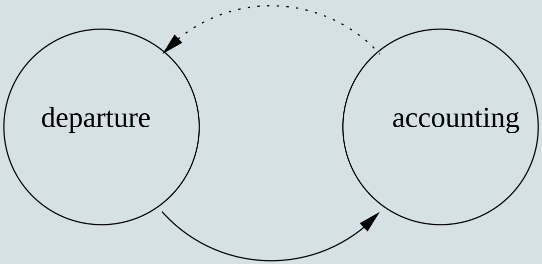
Machine process

Process job (if there is any)



Exit process

Accept completed job and do accounting



The specification language χ:

Modelling and simulation tool for the design of manufacturing systems

The language χ has been developed by the Systems Engineering group

For documentation, see http://se.wtb.tue.nl/documentation

Arrival process

```
type job=real

proc G(a: !job, ta: real) =
|[ u: -> real
 | u:=negexp(ta)
 ; *[ true -> a!time; delta sample u ]
]|
```

Buffer process

```
proc B(a: ?job, b:!job) =
|[ xs: job*, x: job
 | xs:=[]
 ; *[ true;       a?x       -> xs:= xs ++ [x]
    | len(xs)>0; b!hd(xs) -> xs:= tl(xs)
    ]
]|
```

Machine process

```
proc M(a: ?job, b: !job, te: real) =
|[ u: -> real, x: job
 | u:=negexp(te)
 ; *[ true -> a?x; delta sample u; b!x ]
]|
```

Exit process

```
proc E(a: ?job) =
|[ ct,mct: real, n: nat, x: job
 | ct:= 0.0
 ; mct:= 0.0
 ; n:= 0
 ; *[ true -> a?x
            ; ct:= time - x
            ; n:= n + 1
            ; mct:= (n-1)/n*mct + ct/n
            ; !"Mean throughput time ", mct, nl()
    ]
]|
```

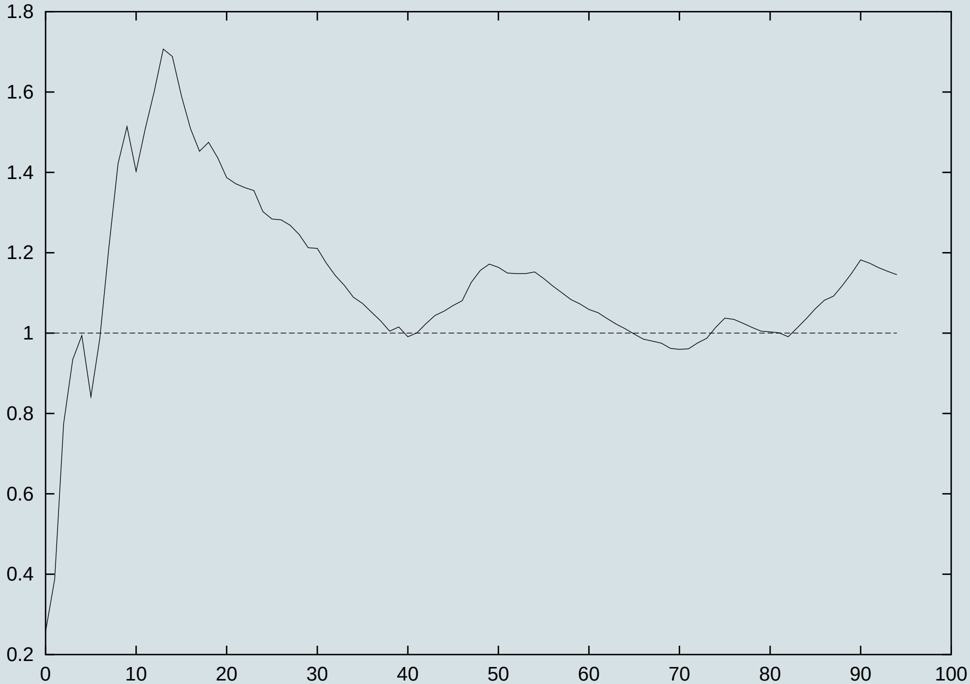
System and simulation experiment

```
clus S() =
|[ a,b,c: -job
 | G(a,1.0) || B(a,b) || M(b,c,0.5) || E(c)
]|

xper = |[ S() ]|
```

Complete χ code

```
from std import *
from random import *

type job=real

proc G(a: !job, ta: real) =
|[ u: -> real
 | u:=negexp(ta)
 ; *[ true -> a!time; delta sample u ]
]|

proc B(a: ?job, b: !job) =
|[ xs: job*, x: job
 | xs:=[]
 ; *[ true;       a?x       -> xs:= xs ++ [x]
    | len(xs)>0; b!hd(xs) -> xs:= tl(xs)
    ]
]|

proc M(a: ?job, b: !job, te: real) =
|[ u: -> real, x: job
 | u:=negexp(te)
 ; *[ true -> a?x; delta sample u; b!x ]
]|
```

```
proc E(a: ?job) =
|[ ct,mct: real, n: nat, x: job
 | ct:= 0.0
 ; mct:= 0.0
 ; n:= 0
 ; *[ true -> a?x
            ; ct:= time - x
            ; n:= n + 1
            ; mct:= (n-1)/n*mct + ct/n
            ; !"Mean throughput time ", mct, nl()
    ]
]|

clus S() =
|[ a,b,c: -job
 | G(a,1.0) || B(a,b) || M(b,c,0.5) || E(c)
]|

xper = |[ S() ]|
```



Mean throughput time as a function of the number of jobs processed for $\lambda = 1$ and $\mu = 2$

More examples...

Other interarrival and service time distributions

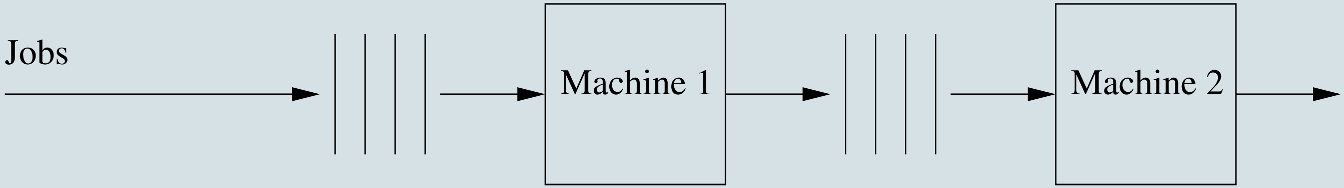
χ has a library available for sampling from distributions, e.g.,

- Bernouilli
- Binomial
- Poisson
- Beta
- Gamma
- Normal
- etc...

Example: Single-stage production system with three parallel machines
In the χ program we have to add channels to the buffer and exit process:

```
proc B(a: ?job, b,c,d: !job) =
|[ xs: job*, x: job
 | xs:=[]
 ; *[ true;       a?x       -> xs:= xs ++ [x]
    | len(xs)>0; b!hd(xs) -> xs:= tl(xs)
    | len(xs)>0; c!hd(xs) -> xs:= tl(xs)
    | len(xs)>0; d!hd(xs) -> xs:= tl(xs)
    ]
]|
proc E(a,b,c: ?job) =
|[ ct,mct: real, n: nat, x: job
 | ct:= 0.0
 ; mct:= 0.0
 ; n:= 0
 ; *[ true -> [ true; a?x -> skip
              | true; b?x -> skip
              | true; c?x -> skip
              ]
            ; ct:= time - x
            ; n:= n + 1
            ; mct:= (n-1)/n*mct + ct/n
            ; !"Mean throughput time ", mct, nl()
    ]
]|
clus S() =
|[ a,b,c,d,e,f,g: -job
 |                              M(b,e,0.5)
 || G(a,1.0) || B(a,b,c,d) || M(c,f,0.5) || E(e,f,g)
 ||                             M(d,g,0.5)
]|
```

Example: Two-stage production system



Jobs are processed by two machines in series. Each machine has its own local buffer and processes jobs in order of arrival. The interarrival and processing times of jobs are exponential with parameters λ, μ_1 and μ_2.

What is the mean (overall) throughput time?

In the χ program we only have to change the system:

```
clus S() =
|[ a,b,c,d,e: -job
 | G(a,1.0) || B(a,b) || M(b,c,0.5) || B(c,d) || M(d,e,0.5) || E(e)
]|
```

The simulation system Arena

In Arena you can construct simulation models without programming, but simply with click, drag and drop...

Student version of Arena is available in the Public Folders in Outlook; look in Software/Overig

Book with CD-ROM:

W. David Kelton, Randall P. Sadowski, Deborah A. Sadowski: Simulation with Arena. 2nd ed., London: McGraw-Hill, 2002

Output analysis of a simulation

Method of independent replications

Example: Long-term ("steady-state") mean waiting time $E(W)$ in the single-stage production line

Produce n *independent* sample paths of waiting times $W_1^{(i)}, W_2^{(i)}, \ldots, W_N^{(i)}$ and compute

$$\bar{W}_N^{(i)} = \frac{1}{N} \sum_{j=1}^{N} W_j^{(i)}, \quad i = 1, \ldots, n.$$

Then, for large N, an approximate $100(1-\delta)\%$ confidence interval for the mean waiting time $E(W)$ is

$$\bar{W}_{n,N} \pm z_{1-\delta/2}\frac{S_{n,N}}{\sqrt{n}}$$

where $\bar{W}_{n,N}$ and $S^2_{n,N}$ are the sample mean and variance of the realizations $\bar{W}^{(1)}_N, \ldots, \bar{W}^{(n)}_N$;

$$\begin{aligned} \bar{W}_{n,N} &= \frac{1}{n}\sum_{i=1}^{n} \bar{W}^{(i)}_N \\ S^2_{n,N} &= \frac{1}{n-1}\sum_{i=1}^{n}(\bar{W}^{(i)}_N - \bar{W}_{n,N})^2 \end{aligned}$$

Results for $\lambda = 0.5$, $\mu = 1$ and 10 runs, each of $N = 10^4$ waiting times

i	$\bar{W}_N^{(i)}$
1	0.995
2	1.002
3	0.959
4	1.037
5	0.902
6	1.011
7	1.125
8	1.007
9	1.075
10	1.044

$E(W) = 1.016 \pm 0.036$ (95% confidence interval)

Results for $\lambda = 0.9$, $\mu = 1$ and 10 runs, each of $N = 10^4$ waiting times

i	$\bar{W}_N^{(i)}$
1	7.373
2	8.496
3	8.574
4	7.752
5	8.637
6	7.404
7	9.556
8	8.863
9	8.537
10	11.000

$E(W) = 8.619 \pm 0.632$ (95% confidence interval)

Clearly, a more congested system is harder to simulate! To obtain a more accurate estimate should we increase the number of runs and/or the length of each run? And, how much?

Problem of the initialization effect

We are interested in the long-term behaviour of the system and maybe the choice of the initial state of the simulation will influence the quality of our estimate.
One way of dealing with this problem is to choose N very large and to neglect this initialization effect. However, a better way is to throw away in each run the first k observations, i.e. we set

$$\bar{W}_N^{(i)} = \frac{1}{N-k} \sum_{j=k+1}^{N} W_j^{(i)}.$$

We call k the length of the *warm-up period* and it can be determined by a graphical procedure.
Disadvantage of the independent replication method is that we have the initialization effect in each simulation run.

Output analysis of a simulation

Batch means

Instead of doing n independent runs, we try to obtain n independent observations by making a *single long run* and, after deleting the first k observations, dividing this run into n subruns.

The advantage is that we have to go through the warm-up period only once.

Let $W_1, W_2, \ldots, W_{nN}$ be the output of a single run, where we have already deleted the first k observations and renumbered the remaining ones. Hence $W_1, W_2, \ldots, W_{nN}$ will be representative for the steady-state. We divide the observations into n batches of length N. Thus, batch 1 consists of

$$W_1, W_2, \ldots, W_N;$$

batch 2 of

$$W_{N+1}, W_{N+2}, \ldots, W_{2N},$$

and so on. Let $\bar{W}_N^{(i)}$ be the sample (or batch) mean of the N observations in batch i, so

$$\bar{W}_N^{(i)} = \frac{1}{N} \sum_{j=(i-1)N+1}^{iN} W_j$$

The $\bar{W}_N^{(i)}$'s play the same role as the ones in the independent replication method. Unfortunately, the $\bar{W}_N^{(i)}$'s will now be *dependent*.

But, under mild conditions, for *large* N the $\bar{W}_N^{(i)}$'s will be approximately independent, each with the same mean $E(W)$.
Hence, for N large enough, it is reasonable to treat the $\bar{W}_N^{(i)}$'s as i.i.d. random variables with mean $E(W)$; thus

$$\bar{W}_{n,N} \pm z_{1-\delta/2}\frac{S_{n,N}}{\sqrt{n}}$$

provides again a $100(1-\delta)\%$ confidence interval for $E(W)$, with $\bar{W}_{n,N}$ and $S^2_{n,N}$ again the sample mean and variance of the realizations $\bar{W}_N^{(1)}, \ldots, \bar{W}_N^{(n)}$; .