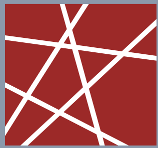


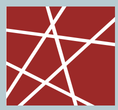
Chair for Network Architectures and Services—Prof. Carle
Department of Computer Science
TU München

Chapter 1: What’s inside a simulator?

Some of today’s slides/figures borrowed from:

Richard Fujimoto
James Kurose, Keith W. Ross
Joachim Warschat
Oliver Rose
Averill Law, David Kelton
Manfred Jobmann

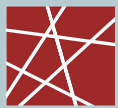
Technische Universität München



Simulation Fundamentals

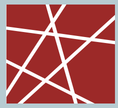
A computer simulation is a computer program that models the behaviour of a physical system over time.

- Program variables (state variables) represent the current state of the physical system.
- Simulation program modifies state variables
  - …to model the evolution of the physical system over time
  - …and/or to incrementally enhance the level of detail of the physical system’s state

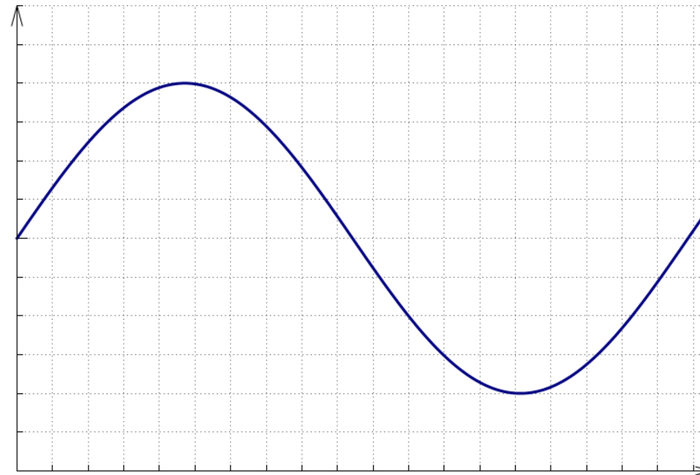


Model taxonomy

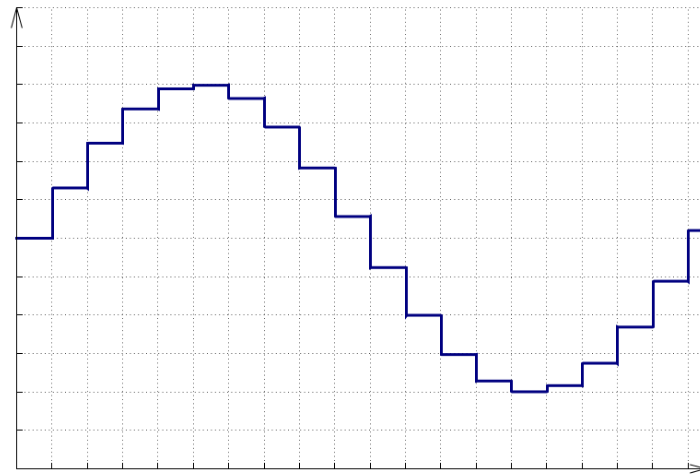
- Static vs. dynamic
  - Static: Simulate state at one point in time / without time
  - Dynamic: State changes over time (focus of lecture!)
- Deterministic vs. stochastic
  - Deterministic: The same input always effects the same output
  - Stochastic: Under same conditions, same input may yield different outputs
    Usual reason: Environment modeled as pseudo-random input
- Continuous vs. discrete
  - cf. next slides …



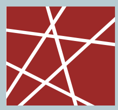
Continuous vs. discrete simulation



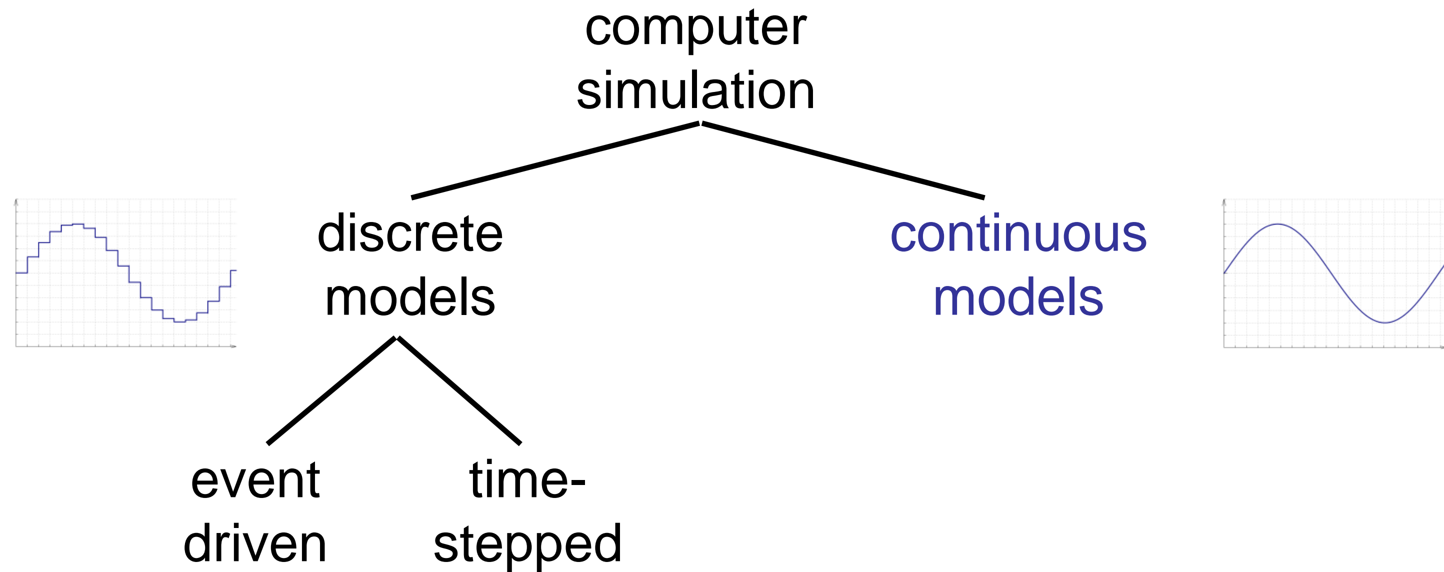
Continuous values



Discrete values

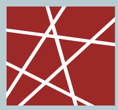


Simulation Taxonomy (1)

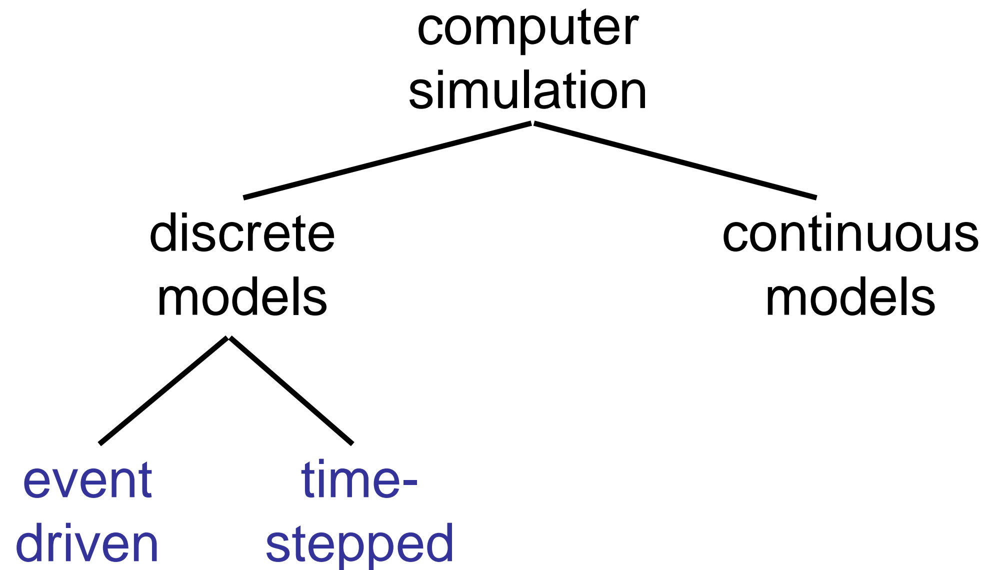


Continuous time simulation

- State changes occur continuously across time
- Typically, behavior described by differential equations
- Example: Flight simulator (time and space are not quantised – at least not at macroscopic dimensions…)

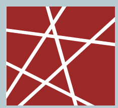


Simulation Taxonomy (2)



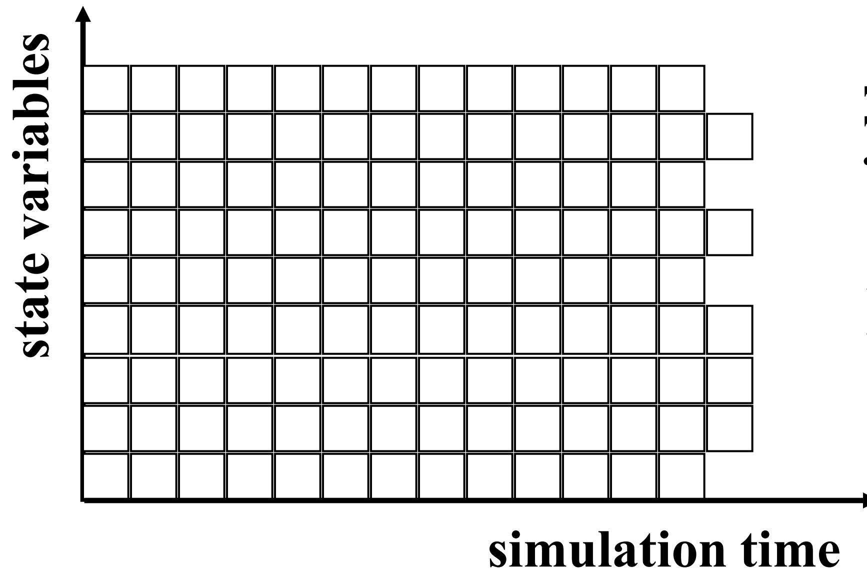
Discrete time simulation [zeitdiskrete Simulation]:

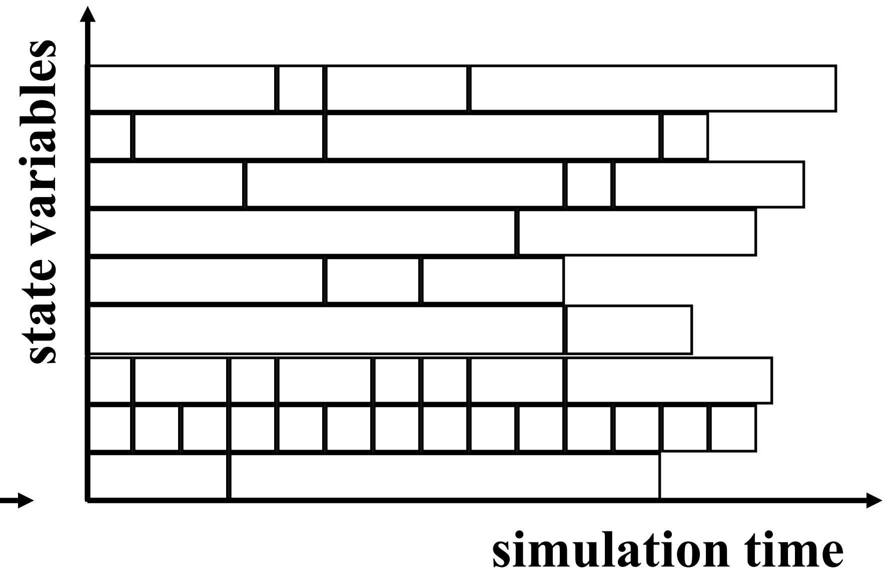
- State changes only occur at discrete time instants
  - Example: Simulating packets in a computer network
- Time stepped: time advances by fixed time increments
- Event stepped: time advances occur with irregular increments, i.e., to the next point "when something happens" (cf. next slide)



Time-stepped vs. event-stepped simulation

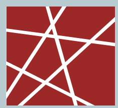
Goal: compute state of system over simulation time





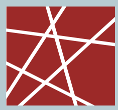
time stepped execution

event driven execution



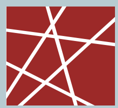
Discrete-event simulation (DES)

- A discrete-event simulation (DES) is the reproduction of the behaviour of a system
  - over time
  - by means of a model where the state variables of the models change immediately at discrete points in time.
- These points in time are the ones in which an event occurs.
- Remark: There are (pseudo-)events that do not lead to changes in the state variables of the model, e.g.:
  - Data collection for statistics / writing to a log file
  - End of simulation
  - Manual garbage collection



What's inside a DES? (1/2: data)

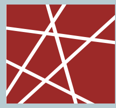
- Simulated time: internal (to simulation program) variable that keeps track of simulated time
  - May progress in huge jumps (e.g., 1ms, then 20s, then 2ms,…)
  - Not related to real time or CPU time in any way!
- System state: variables maintained by simulation program define system state, e.g.: number of packets in queue, current routing table of a router, TCP timeout timers, …
- Events: points in time when system changes state
  - Each event has an associate event time
    - e.g., arrival of packet at a router, departure from the router
    - precisely at these points in time, the simulation must take action (i.e., change state and maybe come up with new future events)
  - Model for time between events (probabilistic) caused by external environment
- Event list: dynamic list of events (→later slides)
- Statistical counters: used for observing the system



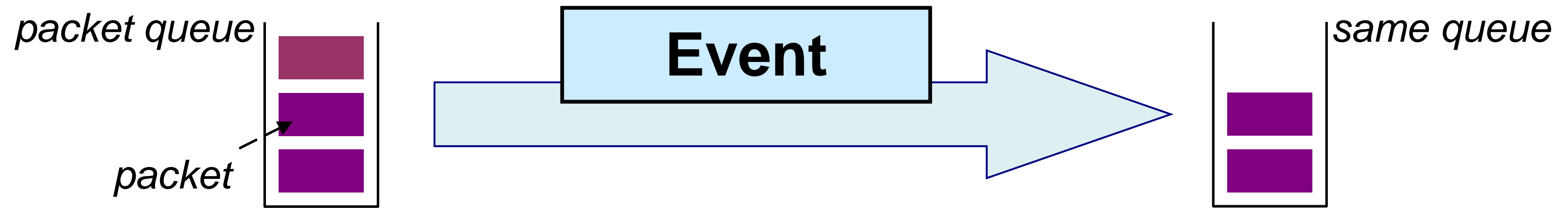
What's inside a DES? (2/2: program code)

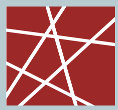
- Timing routine:
  - determines the next event and
  - moves the simulation clock to the next event time
- Event routine: "process the event", i.e., change the system state when an event happens
  - One subroutine per event type
- [P]RNG library routines: generate random numbers
- Report generators: compute performance parameters from statistical counters and generate a report. Runs at simulation end, at interesting events, and/or or at specific pseudo-events
- Main program:

```
while(simulation_time < end_time) {
        next_event = timing_routine();
        next_event.process();
}
```



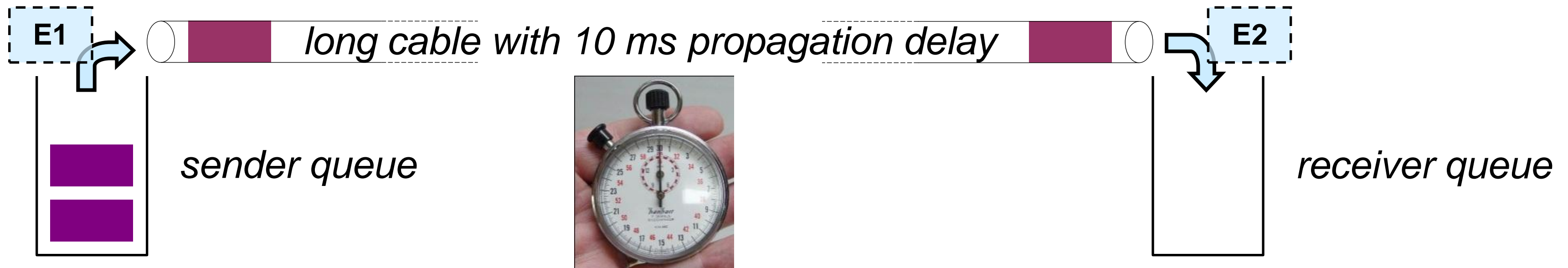
An event changes the state of the model

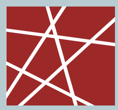




Creating new events

- Not all events are “planned”
- One event may introduce new events in the future:



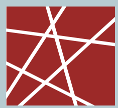


Discrete event simulator: What's inside? (1/2)

Data view:

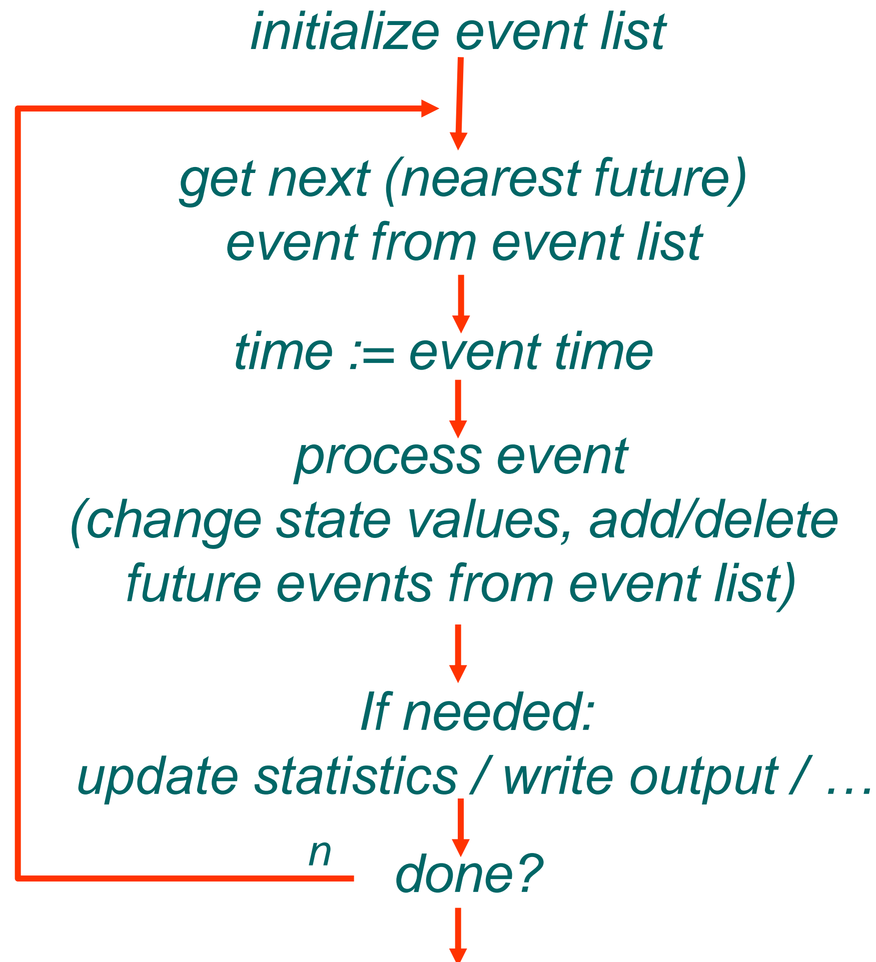
- Simulation program maintains and updates list of future events: event list (sorted!)
- ⇒ Need well-defined set of events
- For each event: simulated system action, updating of event list
- Global variable: Current time (simulated time, not real time or elapsed CPU time!)

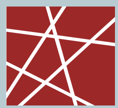
Time	Event
10.0	Take packet P1 out of sender queue and put on link
20.0	Take packet P2 out of sender queue and put on link
27.3	Deliver packet P1 from link to receiver
30.0	Take packet P3 out of sender queue and put on link
37.3	Deliver packet P2 from link to receiver
…	…



Discrete event simulator: What's inside? (2/2)

Control view:

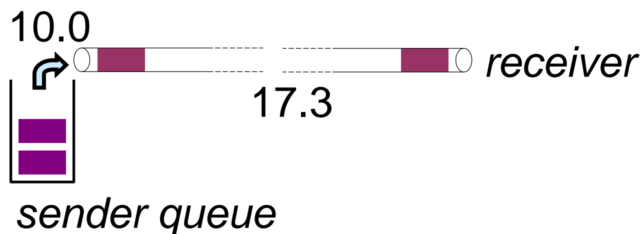


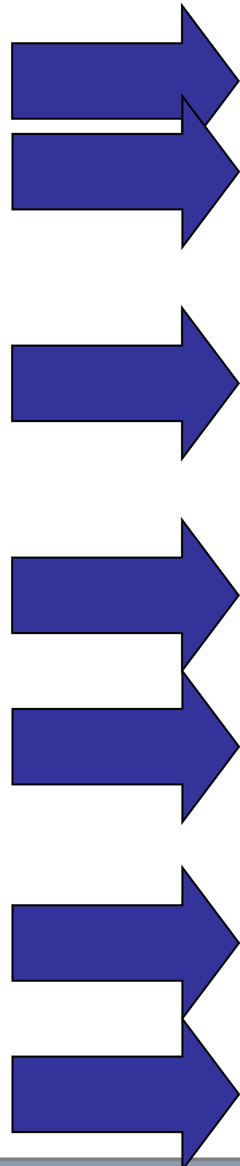


Event list revisited

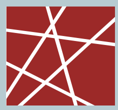
I just lied!

- Example event list was inconsistent: Processing an event may create new events!
- In the example:
  - Transmission delay = 10 time units
  - Propagation delay = 17.3 time units





Time	Event
…	…
10.0	Take packet P1 out of sender queue and put on link
20.0	Take packet P2 out of sender queue and put on link
27.3	Deliver packet P1 from link to receiver
30.0	Take packet P3 out of sender queue and put on link
37.3	Deliver packet P2 from link to receiver
	…

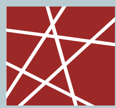


Event list: Data structure

- Must be sorted by event time
- Operations:
  - `insert_event( )`: arbitrary time
  - `get_next_event( )`: newest time
- What kind of data structure to use?
- Answer: Priority queue.

≡ `extract_min( )`

- Algorithms for this data structure (selection):
  - Array or linked list which we keep sorted? — bad idea!
  - Binary heap
  - van Emde Boas tree (vEB tree)
  - Binomial heap
  - …actually, all kinds of search trees that allow efficient execution of `insert( )` and `extract_min( )`

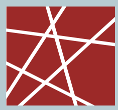


Pure event oriented simulation is difficult

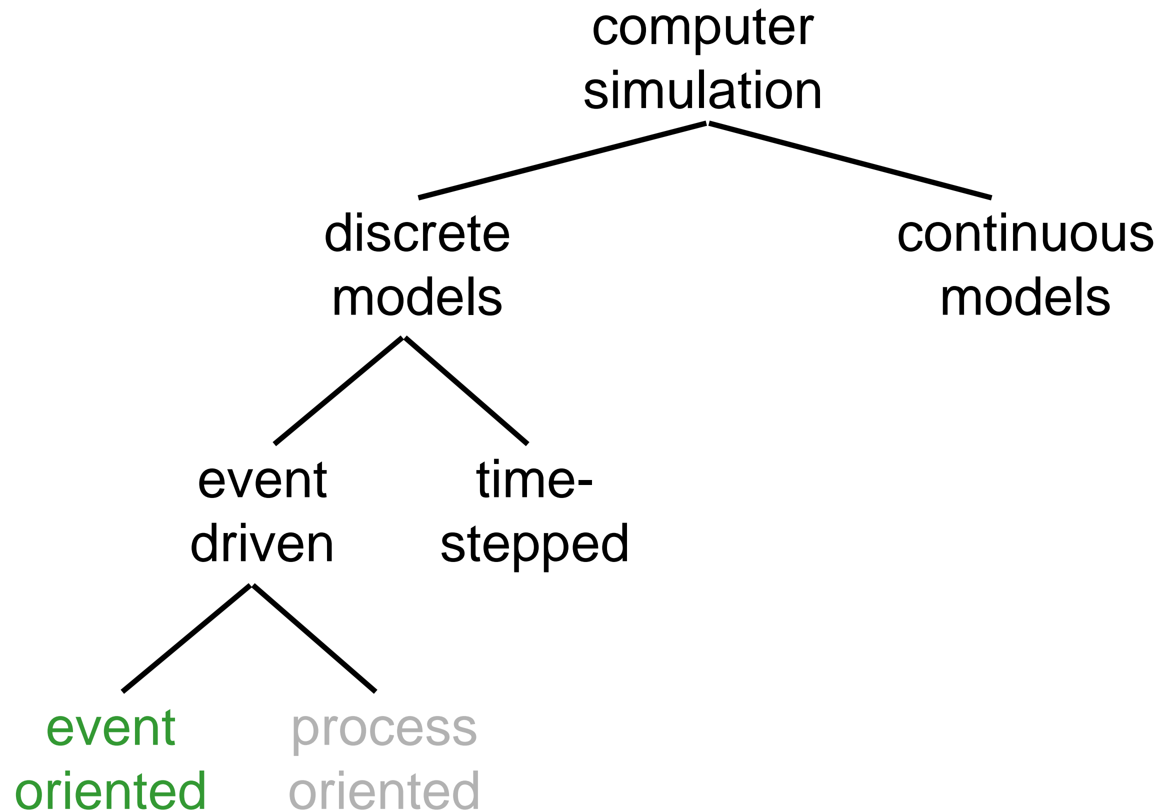
- One event can be a composed of a complicated sequence of many actions:
  - Web client sends HTTP request
  - HTTP request encapsulated into TCP frame
  - TCP frame encapsulated into IP frame
  - IP frame encapsulated into Ethernet frame
  - Put frame into queue of outgoing interface

***one* single event!**
(if we neglect simulating CPU time)

- Even more complicated: Many complicated events (receiving request, sending back answer etc.) that are correlated

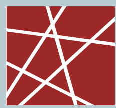


Simulation Taxonomy (3)



Event scheduling approach:

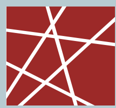
- Event-driven: Simulation as described before
- Focus: on events
- For simple systems



Pure event oriented simulation is difficult

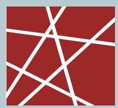
- One event can be a composed of a complicated sequence of many actions:
  - Web client sends HTTP request
  - HTTP request encapsulated into TCP frame
  - TCP frame encapsulated into IP frame
  - IP frame encapsulated into Ethernet frame
  - Put frame into queue of outgoing interface

  one **single event!** (if we neglect simulating CPU time)
- Even more complicated: Many complicated events (receiving request, sending back answer etc.) that are correlated
- Problem #1: Event-based programming doesn't look like normal programming at all!
- Problem #2: Prone to create spaghetti code!?



Solution: Process-oriented simulation

- What is a process? (…in the context of simulation)
  - A body of code
  - Variables allocated to that code
  - Current point of execution in the code
  - ⇒ Not much different from a process in an OS
- How is it used?
  - A process groups sets of related events together
  - A process can execute and then be suspended. Important use cases:
    - Simulation time elapses (e.g., simulate propagation delays)
    - Interactions with other processes that temporarily block (e.g., blocking system calls)
  - Internally, all this is translated into series of events without the programmer noticing it



Applying processes

Two alternative approaches:

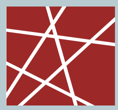
One resource = one process

- Examples: One process for each simulated…:
  - CPU
  - Hard disk
  - Network interface
  - User
- Jobs using these services (e.g., simulated WWW client program)…
  - are data structures
  - are passed from process to process

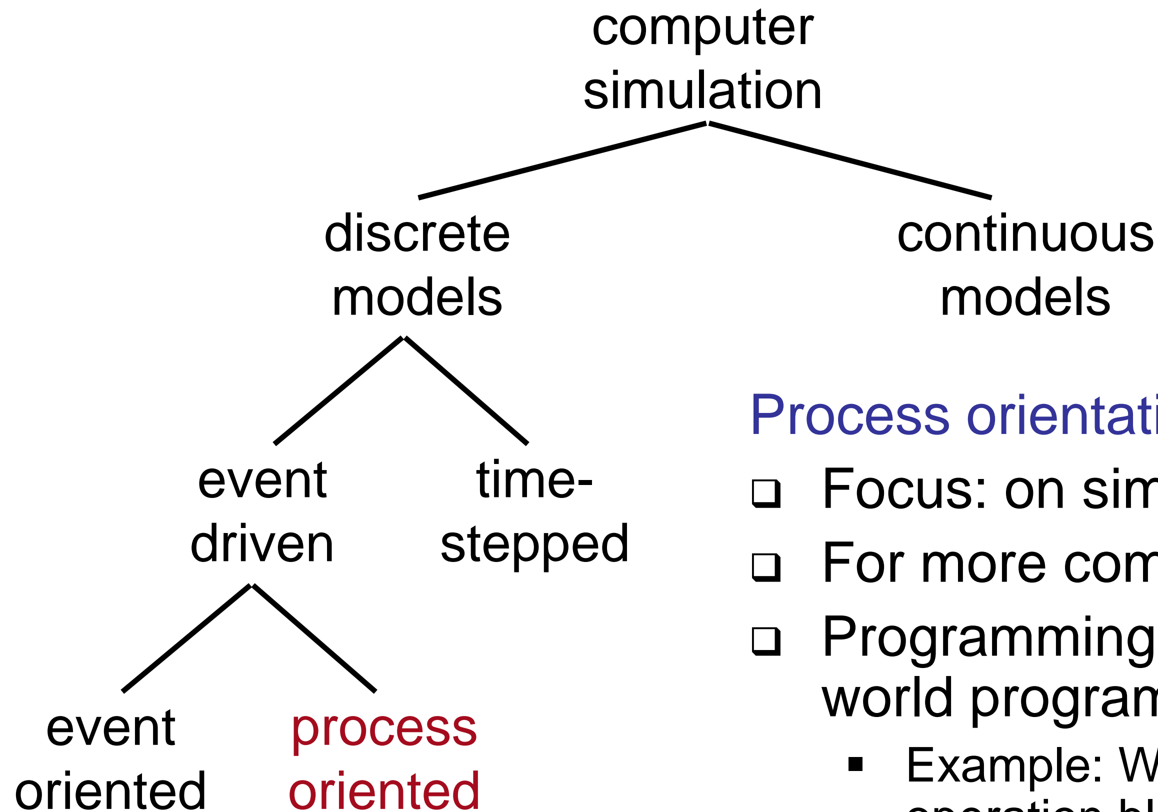
One job = one process

- Examples: One process for each simulated…:
  - WWW client program
  - WWW server program
  - Peer-to-peer client program
- Resources used by these jobs (e.g., simulated network interface)…
  - are global variables / data structures

Which approach is better? — It depends!

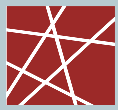


Simulation Taxonomy (4)



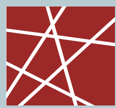
Process orientation:

- Focus: on simulated objects
- For more complex systems
- Programming closer to real-world programming
  - Example: Write into socket; operation blocks
- Usually own simulation language (e.g., OPNet)
- Internally translated into sequence of events



Overview: Event orientation ↔ process orientation

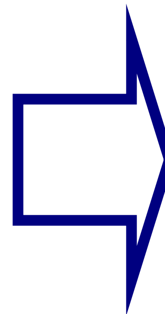
- Event-oriented simulation:
  - Modeler considers one event after the other
  - Simulation clock is stopped during event execution
  - Rather straightforward to implement
  - Often used in non-commercial simulators
- Process-oriented simulation:
  - A process is a ordered series of events related to a certain model object (e.g., customer, job, product)
  - Simulation clock moves on during process execution
  - Commercial simulators use this approach because of simplified model descriptions
  - A process may have several entry points
  - In the simulator kernel, the processes are split into events (may be tricky to implement)



Event list for processes: Usually simpler

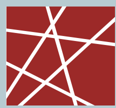
Event-oriented simulation

Time	Event
10.0	Take packet P1 out of sender queue and put on link
20.0	Take packet P2 out of sender queue and put on link
27.3	Deliver packet P1 from link to receiver
30.0	Take packet P3 out of sender queue and put on link
37.3	Deliver packet P2 from link to receiver



Process-oriented simulation (still event-driven!)

Time	Event
10.0	Run queuing process
20.0	Run queuing process
27.3	Run receiver process
30.0	Run queuing process
37.3	Run receiver process

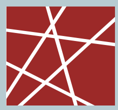


Problem: Simulating blocking behaviour

- Normal programming:

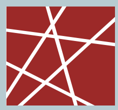
```
result = read(tcp_socket);
// Blocked until tcp_socket has received some data.
use(result);
```

- But how should we simulate the blocking character of `read()` in a process-oriented simulator!?
  - Blocking call: consume simulation time
  - Other events will take place during the time that `read()` is blocked
  - In particular: The event that a new packet has arrived, which in turn triggers the return of the `read()` call!
  - Obviously, these events must not be blocked
  - ⇒Resuming from returning `read()` is a new event



Solutions

- Solution #1: Use threads (??)
  - One thread for process that calls read()
  - One thread for process that moves packet in network
  - One thread for ….
  - Problem: Integration with event concept is difficult
  - Problem: Synchronisation of threads
    - All threads need to access the event list
    - Events must be ordered by time
    - Once some thread has processed an event at time t, then no other thread must generate any event at a time < t
    - [N.B.: Parallel simulation on multiple CPUs is a complex task.]
- Solution #2a: Using continuations
- Solution #2b: Using coroutines



Continuations

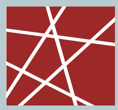
- Normal programming:

```
result = f(parameters);
use(result);
```

- Continuation-passing style:

```
f(parameters, &callback);
// &callback means in C-like syntax:
// pointer or reference to function callback
do_other_stuff(…);

callback(result) {
    // This is just a normal function.
    use(result);
}
```



What is it good for?

- Normal programming:

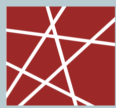
```
result = f(parameters);
// We're blocked until f( ) returns
use(result);
```

- Continuation-passing style:

```
f(parameters, &callback);  //&: pointer
// Will return quickly without blocking.
// Note that f() does not return any results.
maybe_do_other_stuff(…);

f(p, cb) {
        /* Do some calculations; set internal state flags so that
        callback(..) is invoked as soon as the state of the
        current process has been changed by one or more
        events such that we simulate that "f() returns" */
}

callback(result) {
        // A normal function; called to simulate that "f() returns"
        use(result);
}
```



Simulating blocking calls with continuations

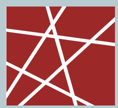
- Normal programming:

```
result = read(tcp_socket);
// Blocked until tcp_socket has received some data.
answer = parse(result);
write(tcp_socket, answer);
```

- Simulator:

```
simulate_read(tcp_socket, &cont);

cont(result) {
    answer = parse(result);
    simulate_write(tcp_socket, answer, &cont2);
}
```



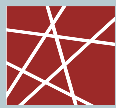
What happens inside of `simulate_read()`?

- `simulate_read(tcp_socket, &cont);`
- Does `simulate_read()` schedule a new event for wakeup with a pointer to `cont()`?
  - No, not quite!
  - When does the data arrive?—We don't know yet!
- Solution: During the processing of this event,…
  - `simulate_read()` passes control to other entities (processes); e.g., reader → IP stack → network card → physical link
  - Each of these entities sets state variables which indicate that new data arriving should wake them up.
  - At some point, the event 'packet received' is processed. The packet gets handled by the various entities (physical link → network card → IP stack → …), and at some point, `cont` gets called, and "`read()` returns"

Problem: Source code difficult to read (1/3)

Normal source code:

```
result = read(socket);
answer = parse(result);
success = write(socket, answer);
if (success == WRITE_OK) {
   blah;
} else if (success == WRITE_FAIL) {
   blubb;
}
```



Problem: Source code difficult to read (2/3)

Simulator code using continuations is one mess of spaghetti code:

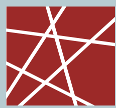
```
read(socket, &cont);

cont(result) {
    answer = parse(result);
    write(socket, answer, &cont2);
}

cont2(result) {
    if (result == WRITE_OK) { …
           … &cont3 …
    } else { …
           … &cont4 …
    }
}

cont3() {
    blah;
}

cont4() {
    blubb;
}
```



Problem: Source code difficult to read (3/3)

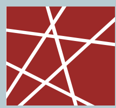
References/pointers to anonymous subroutines/methods help a bit…:

```
read(socket, function(result) { //pointer to anonymous subroutine
    // cont
    answer = parse(result);
    write(socket, answer, function(result2) {
        // cont2
        if (result2 == WRITE_OK) { …
            function(result3) {
                // cont3
                blah;
            }
        } else { …
            function(result4) {
                // cont4
                blubb;
            } // end cont4
        } // end if
    } // end cont2
    ); // end write()
} //end cont
); //end read()
```

… but you'll quickly reach awful levels of indentation!

Coroutines: Generalisation of subroutines

- Subroutine
  - Stateless: local variables always
  - Execution always starts at beginning
  - Execution always ends at last line or `return` statement
  - Returning from subroutine = jump back to calling program context
- Coroutine
  - Can keep state
  - Execution resumes from the place where you left
    (or at the beginning when called for the 1st time)
  - Execution is suspended at `yield` statement (…depends on programming language, of course!) and will resume thereafter
  - Depending on definition/language, yield can specify a target to jump to (i.e., not necessarily the caller of the coroutine!)
  - Multiple coroutines calling and `yield`ing to each other
    ≈ cooperative multitasking



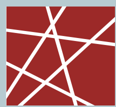
Comparison subroutine—coroutine (1/3)

```
subroutine f() {
    // When called, execution always starts here

    do_stuff;

    return result;
    //Execution always ends at this return statement

    do_more_stuff;
    return something_else;
    //Superfluous lines: these can never be reached!
    //(In Java: "Code unreachable" error)
}
```



Comparison subroutine—coroutine (2/3)

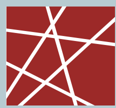
```
coroutine f() {
    // When called for 1st time, execution starts here

    do_stuff;

    yield result;  //Execution suspends here
    // A subsequent call to f() will resume from here!

    do_more_stuff;

    yield something_else;
    // ⇒ f() can return (i.e., yield) a different
    // value upon second call, even if called with same
    // set of parameters and global variables!
}
```



Comparison subroutine—coroutine (3/3)

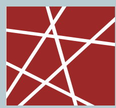
```
coroutine f() {

    do_stuff;

    yield result to somewhere;
    // Suspend execution here and jump to somewhere
    // A subsequent call to f() or yield … to f
    // will resume from here.

    do_more_stuff;

    yield something_else to somewhere_else;
    // ⇒ f() can return (i.e., yield) to a different place.
    // Neither somewhere nor somewhere_else
    // have to be original callers of f()
    // ⇒ A bit like controlled cooperative multitasking
}
```

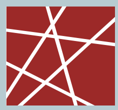


Coroutine support in popular languages

- Native/near-native support for coroutines
  - Simula (cf. next slide)
  - Python, JavaScript 1.7, Ruby
    - Couroutines are called “generators” (Python, JavaScript)
    - Simply use `yield` instead of `return`
  - Perl 6: native support, but rather resembles pipes than subroutines
  - Smalltalk: facilities for fumbling execution stack

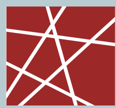
To be fair: Drawing a border line at this point is subjective …

- Some support for coroutines, but not native:
  - C#: some libraries exist; iterators can `yield`
  - Perl 5: a module exists
  - C++: an (unfinished?) library exists
  - Java: emulate with continuations (ugly); library on Sourceforge
  - C: elusive code atrocities required for coroutines, cf. Simon Tatham



Simula—A forgotten curiosity

- Developed in the 1960s (standards: Simula-I, S.-67, S.-87) by Ole-Johan Dahl and Kristen Nygaard (both †2002)
- ≈ Superset of Algol-60
- Purpose: Process-oriented discrete-event simulation
- An underrated pioneering language:
  - The 1st language that introduced coroutines
  - The 1st language that introduced object-oriented programming!
    - Classes
    - Objects
    - Virtual method calls (dynamic binding)
    - Inheritance
    - Some Simula keywords still used today: `class`, `new`, `this`
  - Garbage collection (idea taken from Lisp, 1950s)



Summary of Introduction (Chapters 0 and 1)

- System, model, observer, simulation
- Why and why not to simulate
- Typical workflow / important aspects in a simulation study
  - Verify that your model makes sense
  - Verify the output of your simulation is not just random noise
  - Remember: trash in ⇒ trash out!
- Simulation taxonomy
  - Continuous ↔ discrete
  - Time-based ↔ event-based
  - Event-driven ↔ process-driven
- What's inside a discrete event simulator
  - Event list, sorted by time
  - Simulation time counter
  - State variables
  - Event processing: changes state, but consumes no time
- Continuations and Coroutines