# Modules + Processes

P. Marwedel*
Informatik 12, U. Dortmund

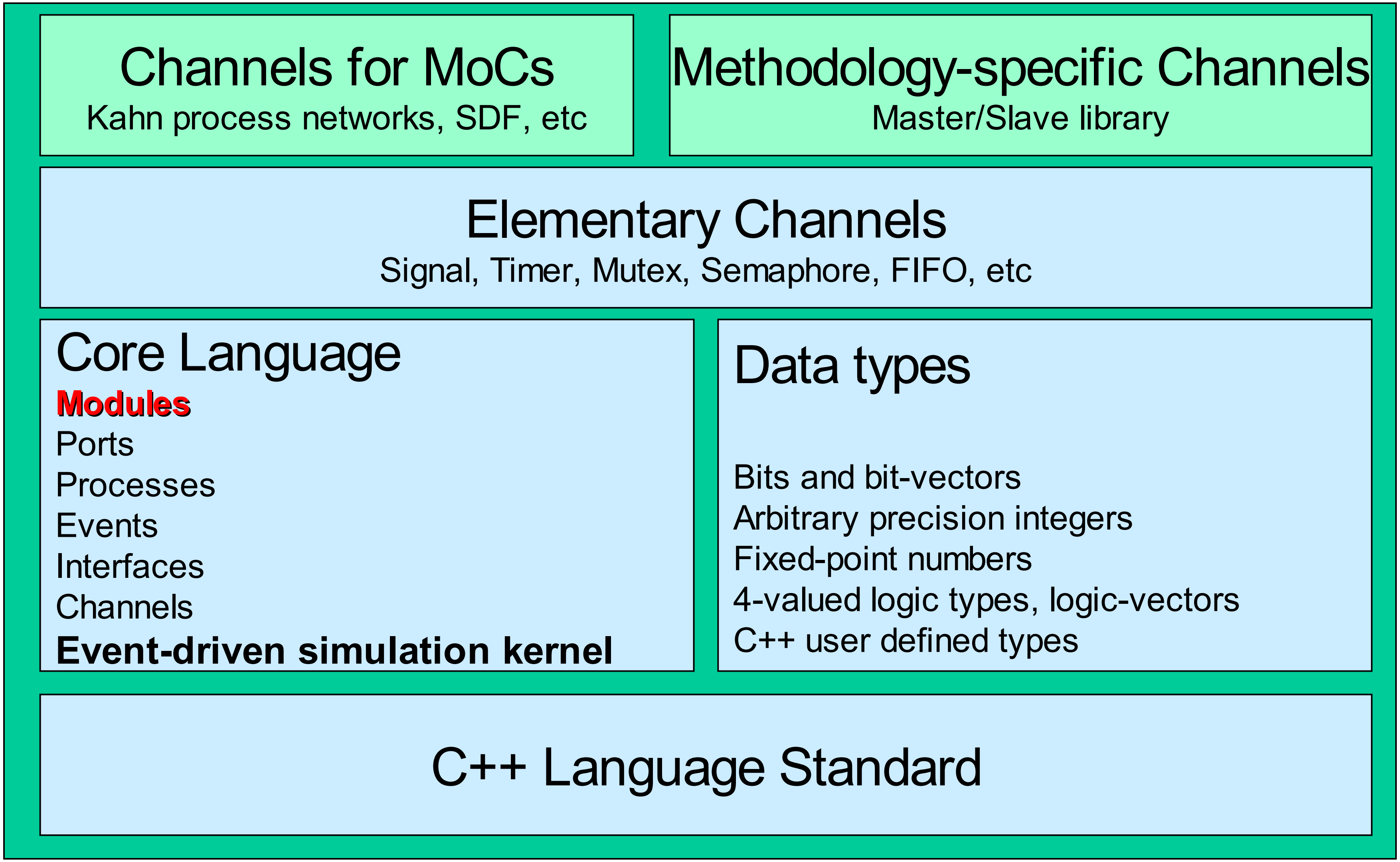
---

* Partially using slides prepared by Tatjana Stankovic from the University of Nis (Serbia and Montenegro), visiting the University of Dortmund under the TEMPUS program.
These slides contain Microsoft cliparts. All usage restrictions apply.

# Contents

- Introduction
- Data types
- A Notion of Time
- ➡ Modules
- Concurrency
- Structure
- Communication, Channels
- Ports & Interfaces
- Advanced Topics

# SystemC language architecture



Channels for MoCs
Kahn process networks, SDF, etc

Methodology-specific Channels
Master/Slave library

Elementary Channels
Signal, Timer, Mutex, Semaphore, FIFO, etc

Core Language
**Modules**
Ports
Processes
Events
Interfaces
Channels
**Event-driven simulation kernel**

Data types
Bits and bit-vectors
Arbitrary precision integers
Fixed-point numbers
4-valued logic types, logic-vectors
C++ user defined types

C++ Language Standard

# Modules

- Modules are basic building blocks of a SystemC design
- A module contains
  - processes (→ functionality)
  - and/or sub-modules (→ hierarchical structure)
- Modules can be
  - described with the **SC_MODULE** macro
  - or derived explicitly from **sc_module**

# Modules

**SC_MODULE** is a macro shorthand for deriving module classes from the library class **sc_module**.

**Syntax:**

**SC_MODULE** (*module_name*) {
 *module_body*

};

**Definition of SC_MODULE:**
**#define SC_MODULE** (*module_name*) \
 **struct** *module_name*: **public sc_module**

# Elements of the module body

```
SC_MODULE( module_name ) {
    // Declaration of module ports
    // Member channel instances
    // Member data instances
    // Member module instances (sub-designs)
    // Constructor
    // Destructor
    // Process member functions (processes)
    // Helper functions
};
```

# The **SC_MODULE** Class Constructor: **SC_CTOR**

A module must contain a C++ constructor.
Constructors should be built using the SystemC macro
**SC_CTOR** or include **SC_HAS_PROCESS** (*module_name*);

**Syntax for SC_CTOR:**

**SC_CTOR**(*module_name*)
: *Initialization* // optional
{ *sub-design allocation*
*sub-design connectivity*
*process_registration*
*miscellaneous setup*
}

The constructor must

- Initialize/allocate sub-designs
- Connect sub-designs
- Declare event sensitivities
- Register processes with the SystemC kernel.

# Contents

- Introduction
- Data types
- A Notion of Time
- Modules
- ➡ Concurrency
- Structure
- Communication, Channels
- Ports & Interfaces
- Advanced Topics

# Processes

**Definition:** A SystemC **process** is a member function or class method of an **SC_MODULE** that is invoked by the scheduler of the SystemC simulation kernel.

**Syntax:**

**void** *process_name*(**void**);

# SC_THREAD

Threads are simple processes created with the **SC_THREAD** macro.

**Syntax:**

**SC_THREAD**(*process_name*);

Registration by using the macro within the constructor.

**Context:**

**SC_MODULE**(*simple_process*){

 **SC_CTOR**(*simple_process*) {

  **SC_THREAD**(*thread_process*);

  }

  **void** *thread_process*(**void**);

};

# SC_THREAD

Processes created using **SC_THREAD**

- are started *once*,
- execute until they call **wait**() or **return**, (**wait**() may be called indirectly via blocking read or write calls)
- frequently comprise infinite loops including **wait** calls,
- provide **persistence** for all local variables,
- roughly correspond to processes in an operating system using "cooperative multitasking“ or coroutines.

# SC_THREAD::wait()

**Examples:**

**wait**(*time*);

**wait**(*event*);

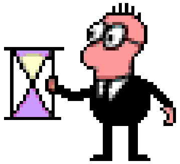
**wait**(*event* [, | *event*]); // any event causes resume

**wait**(*event* [, & *event*]); // all events req. for resume

**wait**(*timeout*, *event*); // event with timeout

**wait**(); // use static sensivity

Function **time_out**() may be called immediately after a wait to check if **wait** was terminated by timeout.



# SC_METHOD

Macro **SC_METHOD** can be used to create a process within a constructor. Such a process

- *Always runs to completion*;
- Cannot call **wait()** or blocking **read** and **write** methods.
- Is more efficient than processes created with **SC_THREAD**.
- Looses the values of all variables upon exit.

# next_trigger()

The **next_trigger()** method can be used to specify the sensivity of **SC_METHOD**s. The last call of **next_trigger** preceding an exit from an **SC_METHOD**s specifies the condition for re-entry.

**Example:**

**next_trigger**(*time*);

**next_trigger**(*event* [, | *event*]); // any event causes restart

**next_trigger**(*event* [, & *event*]); // all events req. for restart

**next_trigger**(*timeout*, *event*); // event with timeout

**next_trigger**(); // back to static sensivity

# Static sensivity

***Static sensivity*** is an alternative to using **wai**t() or **next_ trigger**(). Static sensivity is established at compile time.

Static sensivity applies to the most recent process registration.

2 syntactical forms:

1. **Streaming style**:
   **sensitive** << *event* [ <<*event* ] …;
2. **"Functional style"**
   **sensitive** (*event* [, *event*] ..);

# don't_initialize

Normally, all processes are started initially.
This can be avoided by calling **dont_initialize**().
The call must follow process registration.
A static sensivity list should be used with **dont_initialize,**
otherwise the process would never be executed.

Example:

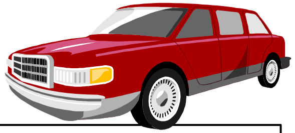
```
…
SC_METHOD(attendant_method);
  sensitive(fillup_request);
  dont_initialize();
…
```

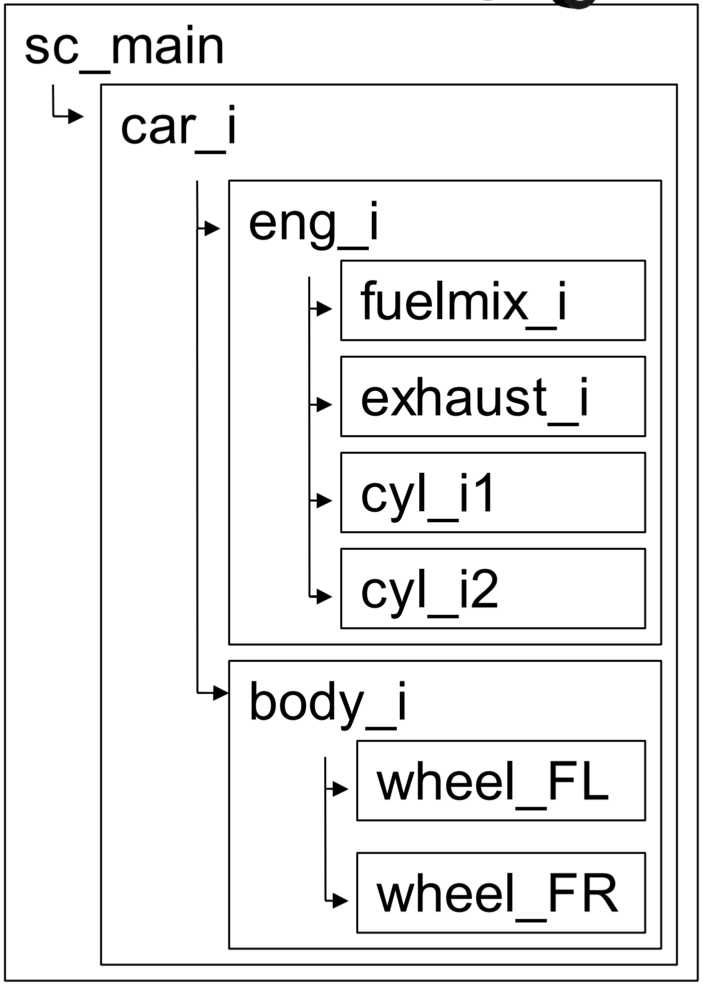
# Contents

- Introduction
- Data types
- A Notion of Time
- Modules
- Concurrency
- ➡ Structure
- Communication, Channels
- Ports & Interfaces
- Advanced Topics

# Design Hierarchy

## Example





- sc_main
  - car_i
    - eng_i
      - fuelmix_i
      - exhaust_i
      - cyl_i1
      - cyl_i2
    - body_i
      - wheel_FL
      - wheel_FR

Different module instantiation approaches:

- in the header or the implementation file,
- direct instantiation in declaration or dynamic creation with pointers,
- at top level of the hierarchy or at some other level.

Combinations thereof.

# Direct Top-Level Instantiation

**Example:**

```
//file: main.cpp
#include "Wheel.h"
int sc_main (int argc, char* argv[] {
Wheel wheel_FL("wheel_FL");
Wheel wheel_FR("wheel_FR");
sc_start();
}
```

Simple and to the point.

# Indirect Top-Level Instantiation

**Example:**

```
//file: main.cpp
#include "Wheel.h"
int sc_main (int argc, char* argv[] {
  Wheel* wheel_FL; // pointer to FL wheel
  Wheel* wheel_FR; // pointer to FR wheel
  wheel_FL = new Wheel("wheel_FL"); //create FL
  wheel_FR = new Wheel("wheel_FR"); //create FR
  sc_start();
  delete wheel_FL;
  delete wheel_FR;
}
```

More code, but more flexible: looping, arrays, if then etc. possible.

# Direct Sub-Module Header-Only Instantiation

**Example:**

```
//file: Body.h
#include "Wheel.h"
SC_MODULE(Body)  {
  Wheel wheel_FL;
  Wheel wheel_FR;
  SC_CTOR(Body)
  : wheel_FL("wheel_FL"),  //initialization
    wheel_FR("wheel_FR")  //initialization
  {
   // other initialization
  }
}
```

Initializer list to be used due to C++ rules

# Indirect Sub-Module Header-Only Instantiation

**Example:**

```
//file: Body.h
#include "Wheel.h"
SC_MODULE(Body)  {
  Wheel* wheel_FL;
  Wheel* wheel_FR;
  SC_CTOR(Body) {
   wheel_FL = new Wheel("wheel_FL");
   wheel_FR = new Wheel("wheel_FR");
   // other initialization
  }
};
```

Easier to read than direct method.

# Direct Sub-Module Instantiation

```
//file: Body.h
#include "Wheel.h"
SC_MODULE(Body) {
  Wheel wheel_FL;
  Wheel wheel_FR;
  SC_HAS_PROCESS(Body);
  Body(sc_module_name nm);
};
```

Moves the complexity of initialization to the implementation, hides it from the users of the header file.

```
//file: Body.cpp
#include "Body.h" // Constructor:
Body::Body(sc_module_name nm)
: wheel_FL("wheel_FL"),
  wheel_FR("wheel_FR"),
  sc_module(nm)
{ .. other initialization ..}
```

# Indirect Sub-Module Instantiation

```
//file: Body.h
struct Wheel;
SC_MODULE(Body) {
  Wheel* wheel_FL;
  Wheel* wheel_FL;
  SC_HAS_PROCESS(Body);
  Body(sc_module_name nm);
};
```

```
//file: Body.cpp
#include "Wheel.h" // Constructor:
Body::Body(sc_module_name nm)
: sc_module(nm)
{wheel_FL=new Wheel("wheel_FL");
 wheel_FR=new Wheel("wheel_FR");
 .. other initialization .. }
```

Moves the complexity of initialization to the implementation, hides it from the users of the header file.

# Contrasting Implementation Approaches

| **Level** | **Allocation** | **Pros** | **Cons** |
|---|---|---|---|
| Main | Direct | Least code | Inconsistent with other levels |
| Main | Indirect | Dynamically configurable | Involves pointers |
| Module | Direct<br>Header only | All in one file, easier to understand | Requires sub-module headers |
| Module | Indirect<br>Header only | All in one file, dynamically configurable | Involves pointers |
| Module | Direct with separate compilation | Hides implementation | Requires sub-module headers |
| Module | Indirect with separate compilation | Hides sub-module headers and implementation, dynamically configurable | |

# Hello World in SystemC

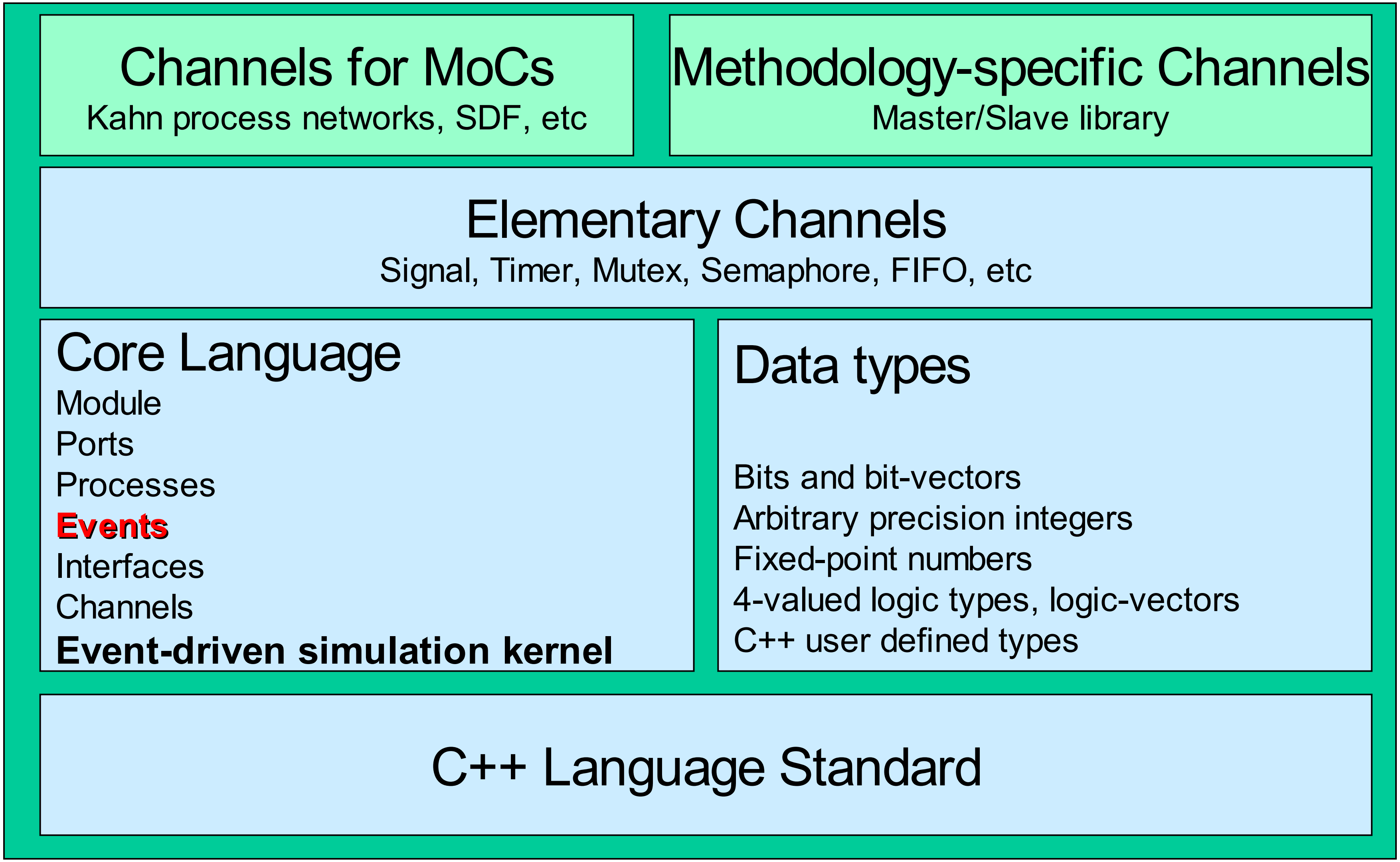
```
//FILE:main.cpp
#include <Hello_SystemC.h>
int sc_main(int argc, char* argv[]) {
  const sc_time t_PERIOD(8,SC_NS);
  sc_clock clk("clk",t_PERIOD);
  Hello_SystemC iHello_SystemC
   ("iHello_SystemC");
  iHello_SystemC.clk_pi(clk);
  sc_start(10);
  return 0; }
```

```
//FILE:Hello_SystemC.cpp
#include <Hello_SystemC.h>
void Hello_SystemC::main_method(void)
 {std::cout << sc_time_stamp() <<
  " Hello world!" << std::endl;
}
```

```
#ifndef HELLO_SYSTEMC_H
#define HELLO_SYSTEMC_H
//FILE:Hello_SystemC.h
#include <systemc.h>
#include <iostream>
SC_MODULE(Hello_SystemC) {
  sc_in_clk clk_pi;
  void
Hello_SystemC::main_method(void);
  SC_CTOR(Hello_SystemC) {
    SC_METHOD(main_method);
    sensitive << clk_pi.neg();
    dont_initialize();
  } };
#endif
```

Source & ©: D. Black, J. Donovan, http://eklectically.com/Book/; All usage restrictions imposed by the authors apply.

# SystemC language architecture



Channels for MoCs

Kahn process networks, SDF, etc

Methodology-specific Channels

Master/Slave library

Elementary Channels

Signal, Timer, Mutex, Semaphore, FIFO, etc

Core Language

Module
Ports
Processes
**Events**
Interfaces
Channels
**Event-driven simulation kernel**

Data types

Bits and bit-vectors
Arbitrary precision integers
Fixed-point numbers
4-valued logic types, logic-vectors
C++ user defined types

C++ Language Standard

# Contents

- Introduction
- Data types
- A Notion of Time
- Modules
- Concurrency
- Structure
- ➡ Communication
- Channels
- Advanced Topics

# Events

An event is an object represented by **sc_event**s, that determines whether and when a process‘s execution should be triggered or resumed.

An event is used to represent a condition that may occur during the course of simulation and to control the triggering of processes accordingly.

The **sc_event** class provides basic synchronization for processes.

# Events

Distinction between

- event (“subjunctive mood“) and
- actual occurrence of an event (“indicative mood“).

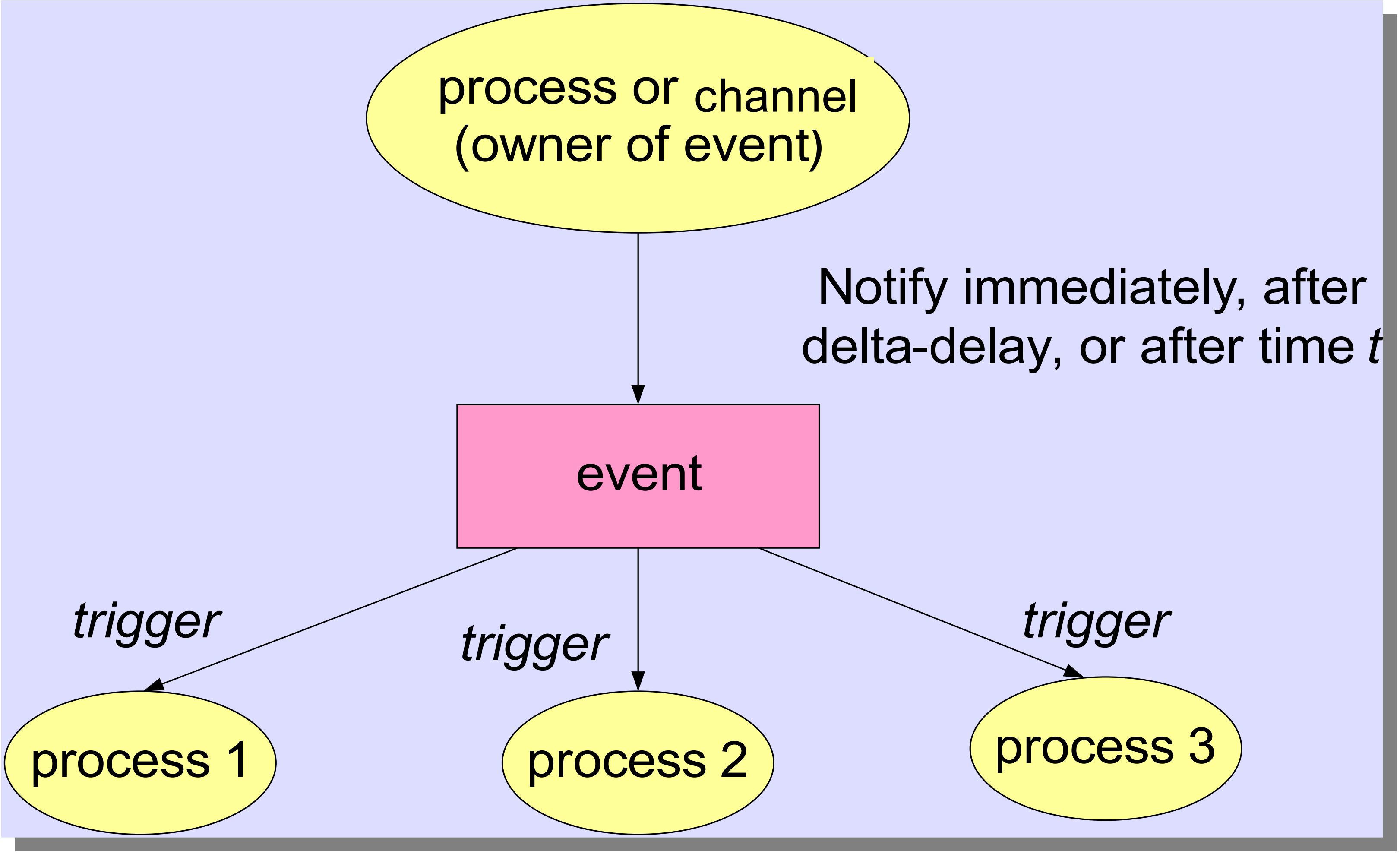
An event is usually associated with some change of state in a process or of a channel.

The owner of the event is responsible for reporting the change to the event object.

The act of reporting the change to the event is called **notification.**

# Event

Event notification and process triggering


process or channel
(owner of event)

Notify immediately, after delta-delay, or after time *t*

event

*trigger*

*trigger*

*trigger*

process 1

process 2

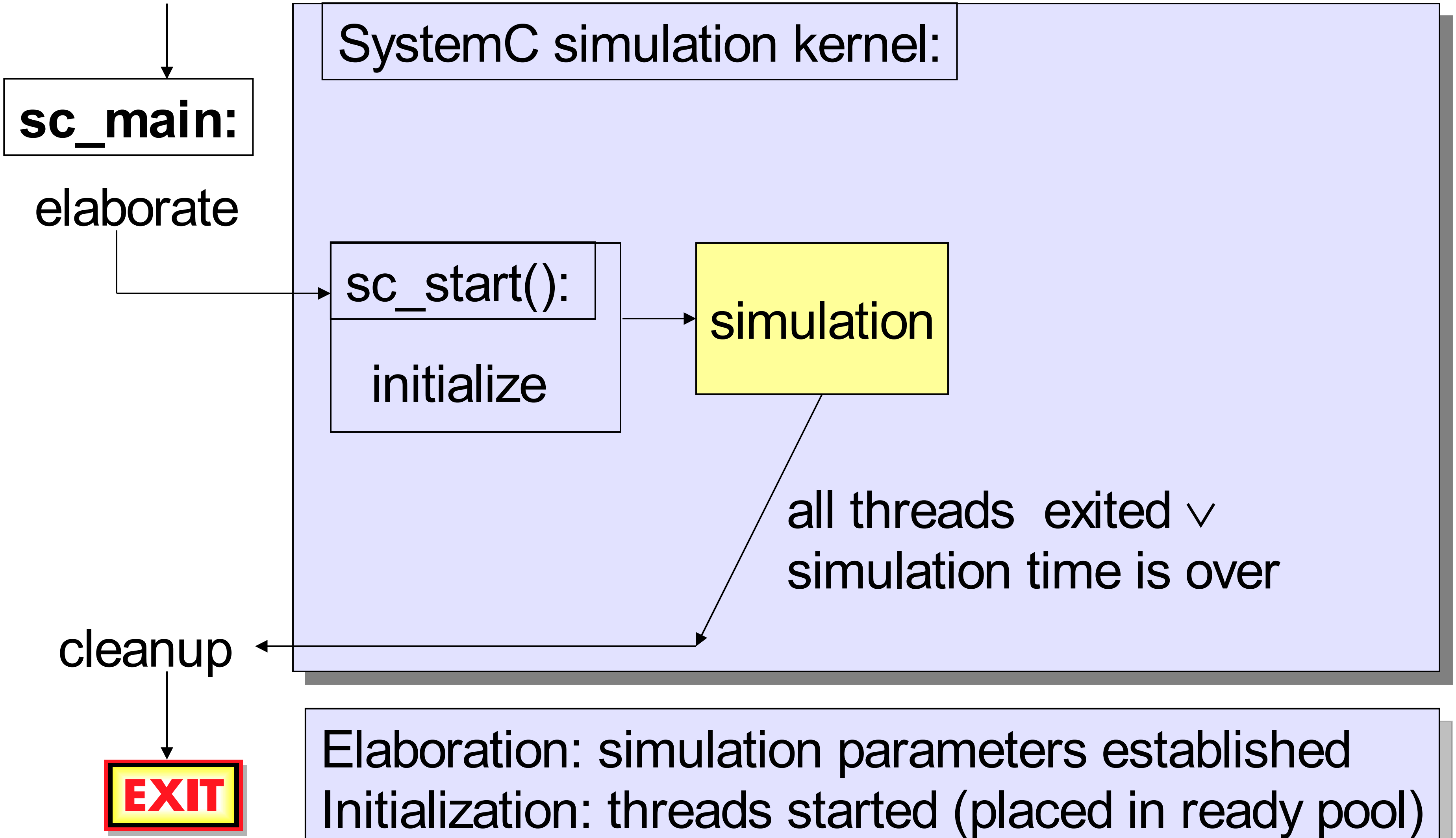
process 3

# Triggering events: .notify

Events are created with **.notify().**

- **Immediate notify's** move waiting processes immediately from the waiting pool to the ready pool.
  Examples:
  *event_name*.**notify**(); **notify**(*event_name);*
- **Delayed notification:**
  *event_name*.**notify**(**SC_ZERO_TIME**);
  **notify**(*event_name*, **SC_ZERO_TIME**);
  Processes waiting will execute at the next δ cycle.
- **Timed notification** lets processes execute after the indicated time.
  *event_name*.**notify**(t); **notify**(*event_name,* t);

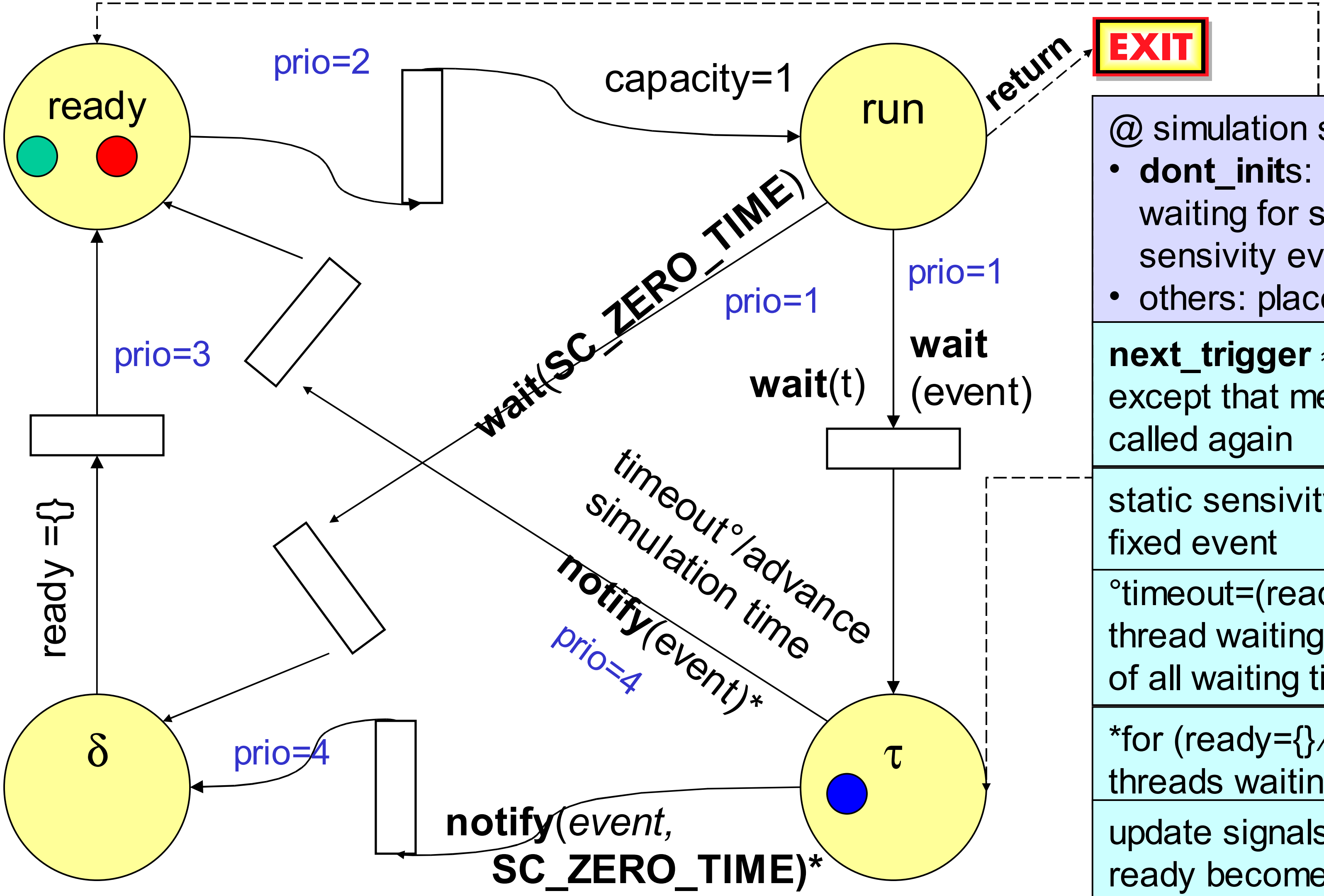
# Simulation engine (simplified)


sc_main:
elaborate
SystemC simulation kernel:
sc_start():
initialize
simulation
all threads exited $\vee$ simulation time is over
cleanup
EXIT

Elaboration: simulation parameters established
Initialization: threads started (placed in ready pool)

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -



@ simulation start:

- **dont_init**s: placed in $\tau$ waiting for static sensivity event
- others: placed in ready

**next_trigger** ≈ **notify**, except that methods are called again

static sensivity ≈ **wait** for fixed event

°timeout=$(\text{ready}=\{\}\wedge\delta=\{\})\wedge$ thread waiting time = min. of all waiting times

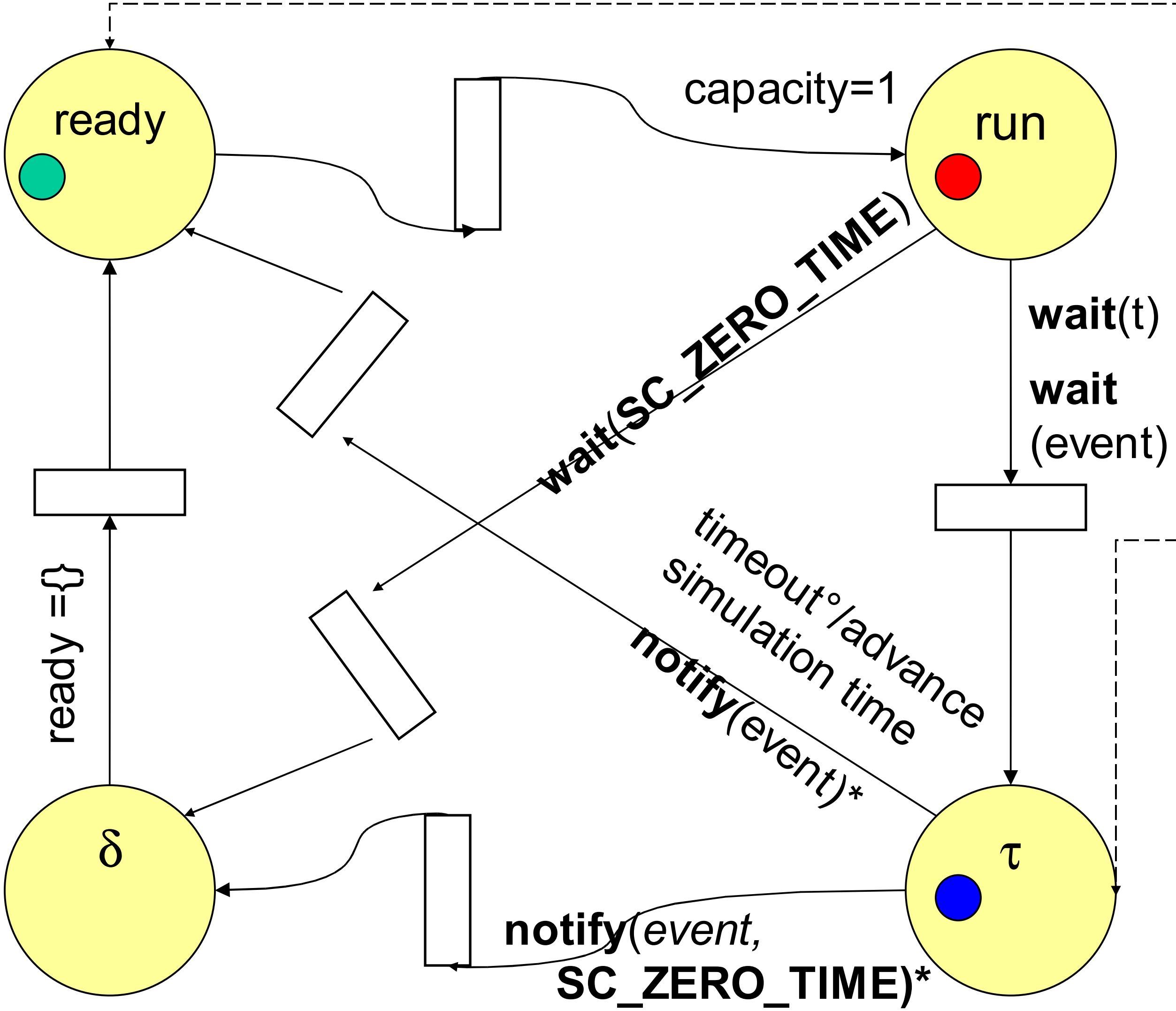
*for $(\text{ready}=\{\}\wedge\delta=\{\})$: for threads waiting for event

update signals each time ready becomes {} (end of $\delta$ cycle), see **sc_signal**

diagram not found elsewhere

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -

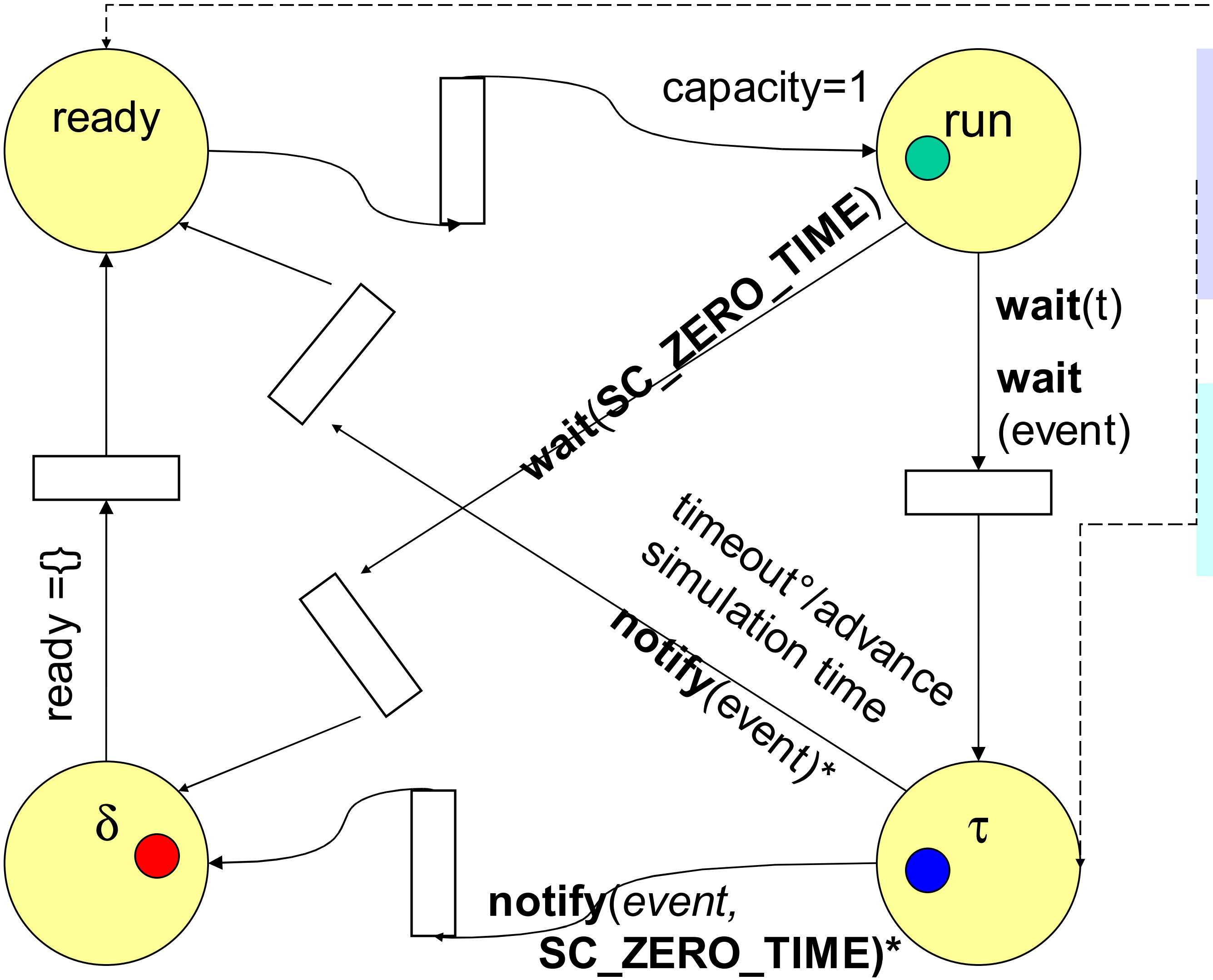


@ simulation start:
- **dont_inits**: placed in τ waiting for static sensivity event
- others: placed in ready

non-deterministic choice between ready threads

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -



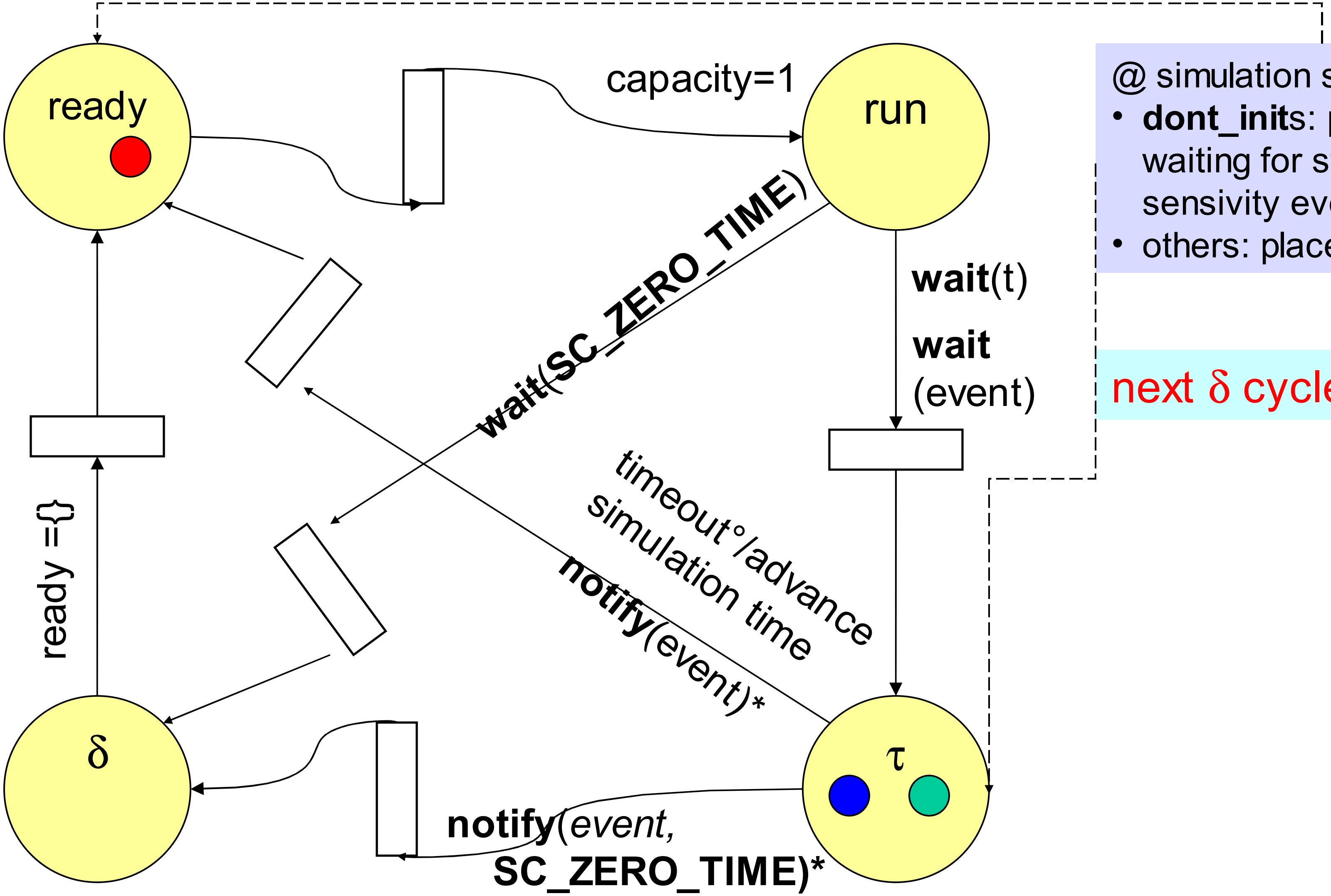
@ simulation start:

- **dont_inits**: placed in τ waiting for static sensivity event
- others: placed in ready

ready pool emptied, before next δ cycle starts

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -



ready

capacity=1

run

**wait**(SC_ZERO_TIME)

**wait**(t)

**wait** (event)

timeout°/advance simulation time

**notify**(event)*

ready ={}

δ

τ

**notify**(*event,* **SC_ZERO_TIME)***

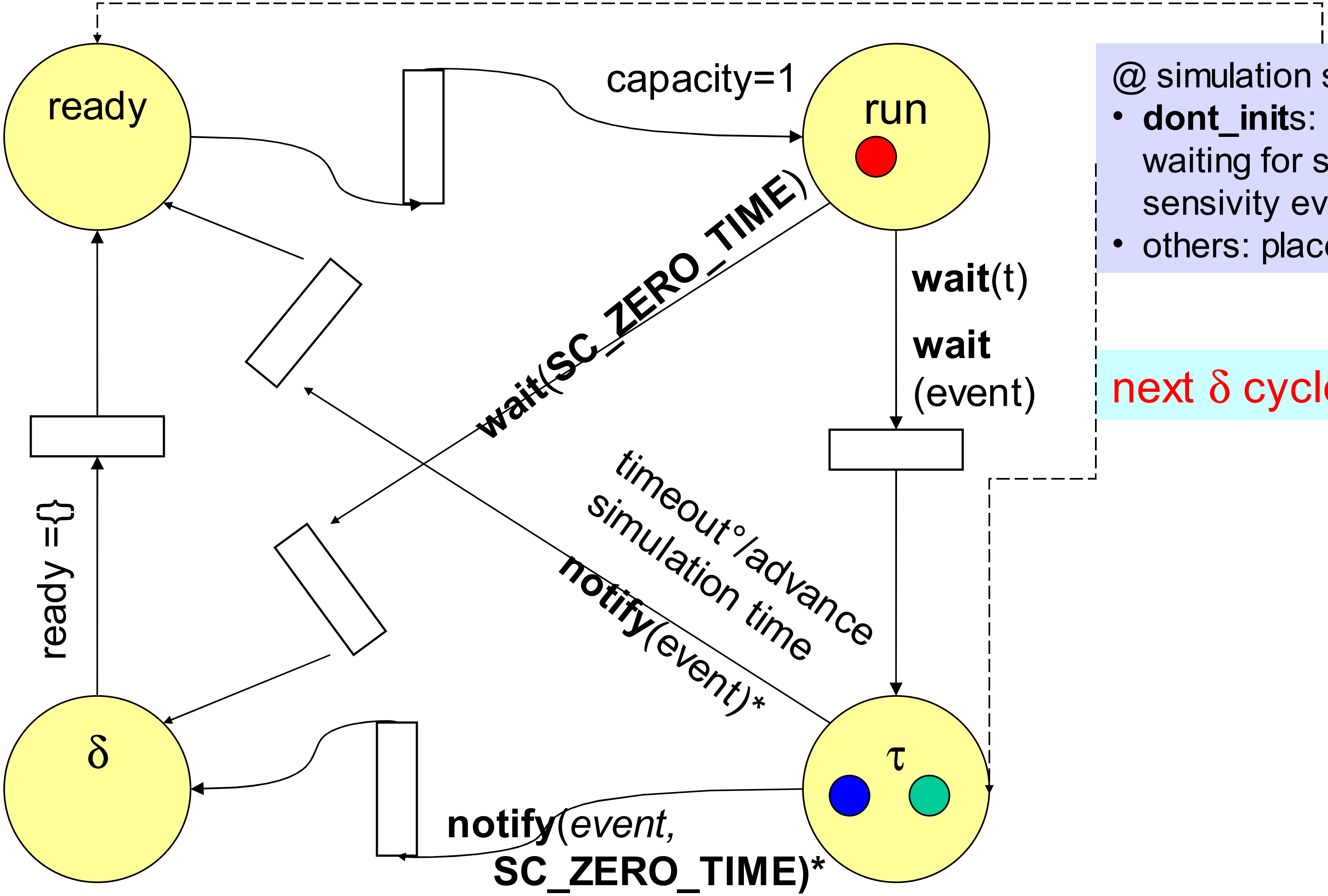
@ simulation start:

- **dont_inits**: placed in τ waiting for static sensivity event
- others: placed in ready

next δ cycle

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -



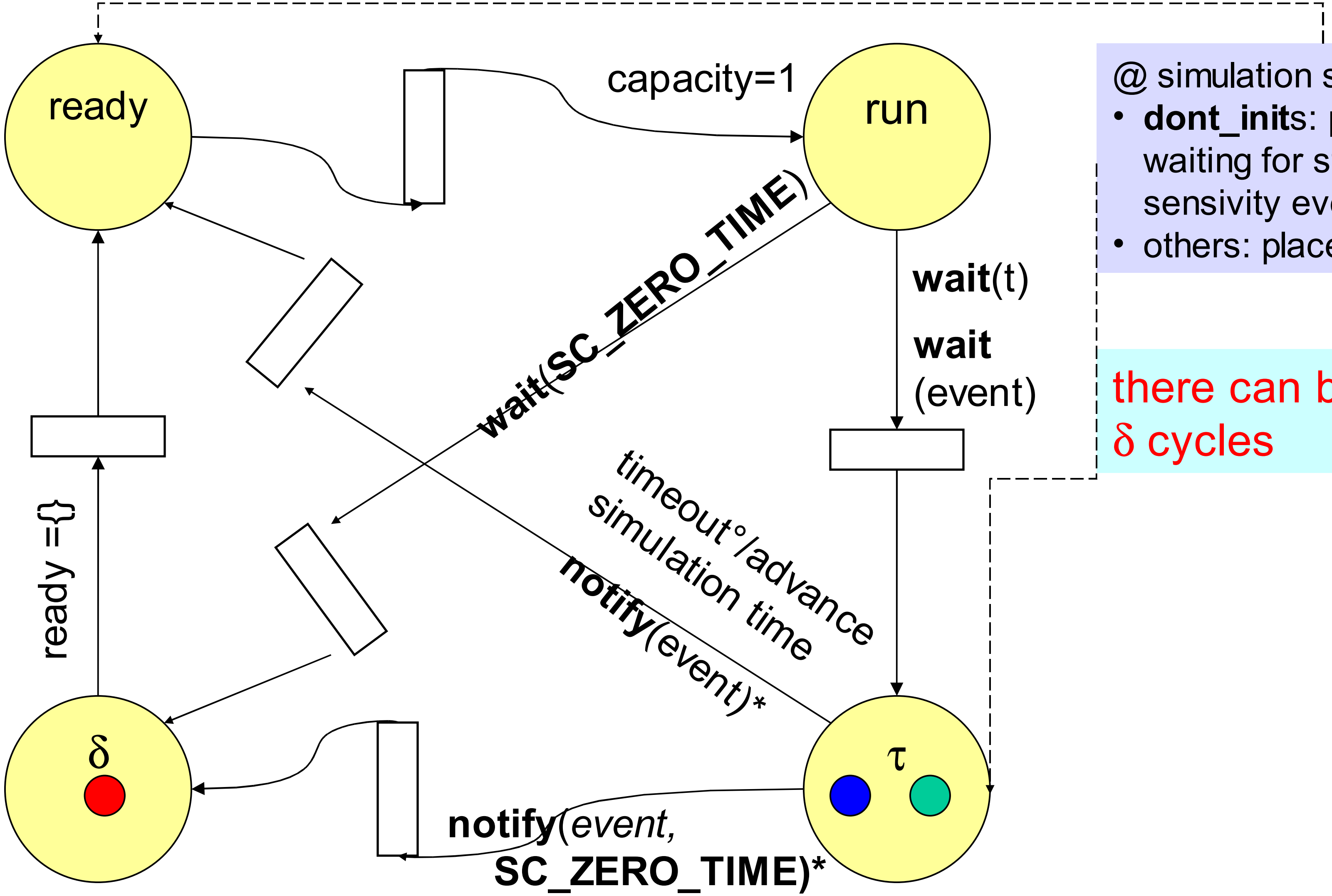
@ simulation start:

- **dont_init**s: placed in τ waiting for static sensivity event
- others: placed in ready

next δ cycle …

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -



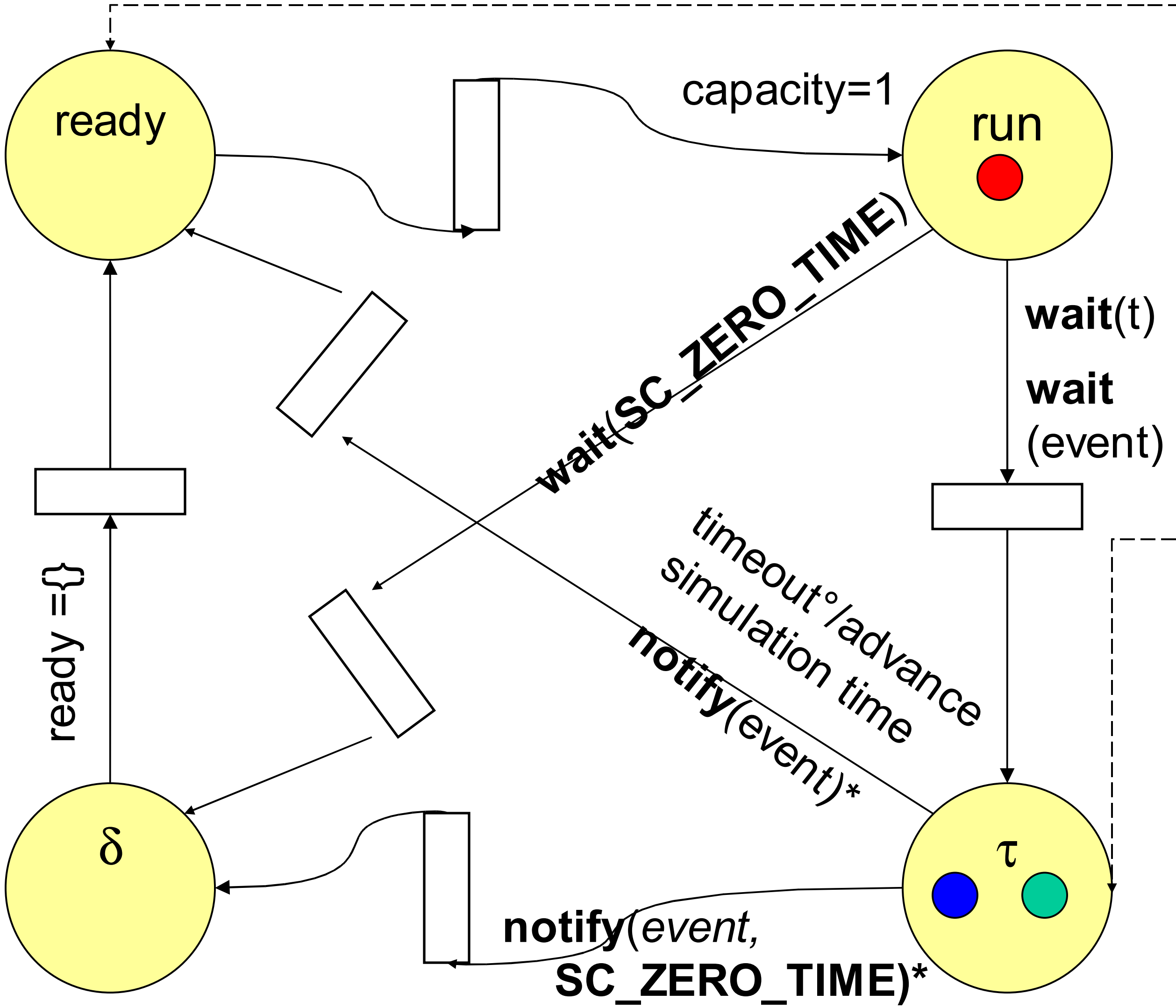
@ simulation start:

- **dont_init**s: placed in τ waiting for static sensivity event
- others: placed in ready

there can be several δ cycles

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -



ready

run

capacity=1

**wait**(SC_ZERO_TIME)

**wait**(t)

**wait** (event)

timeout°/advance simulation time

**notify**(event)*

ready ={}

δ

τ

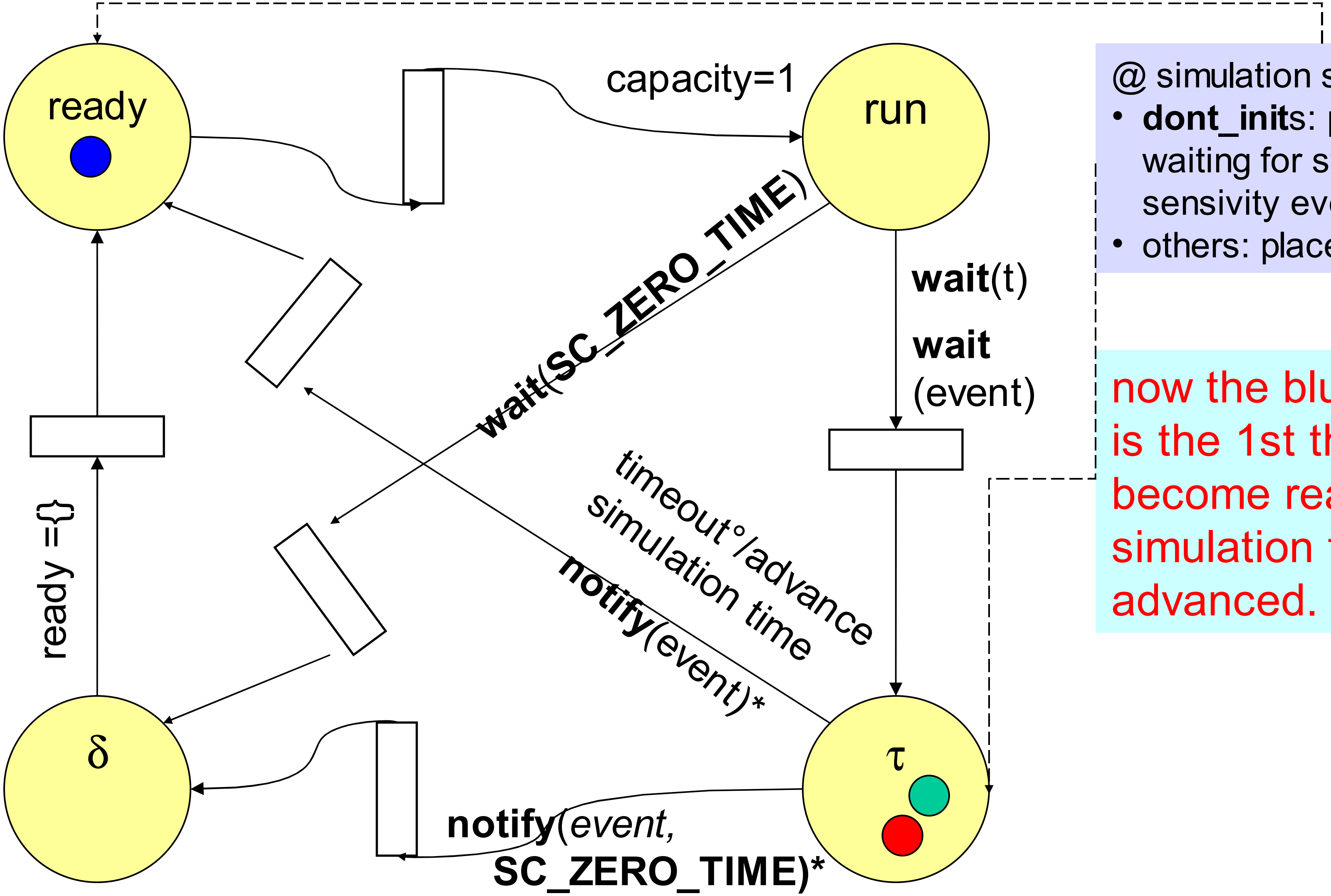
**notify**(*event,* **SC_ZERO_TIME)***

@ simulation start:
- **dont_init**s: placed in τ waiting for static sensivity event
- others: placed in ready

there can be several δ cycles …

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -



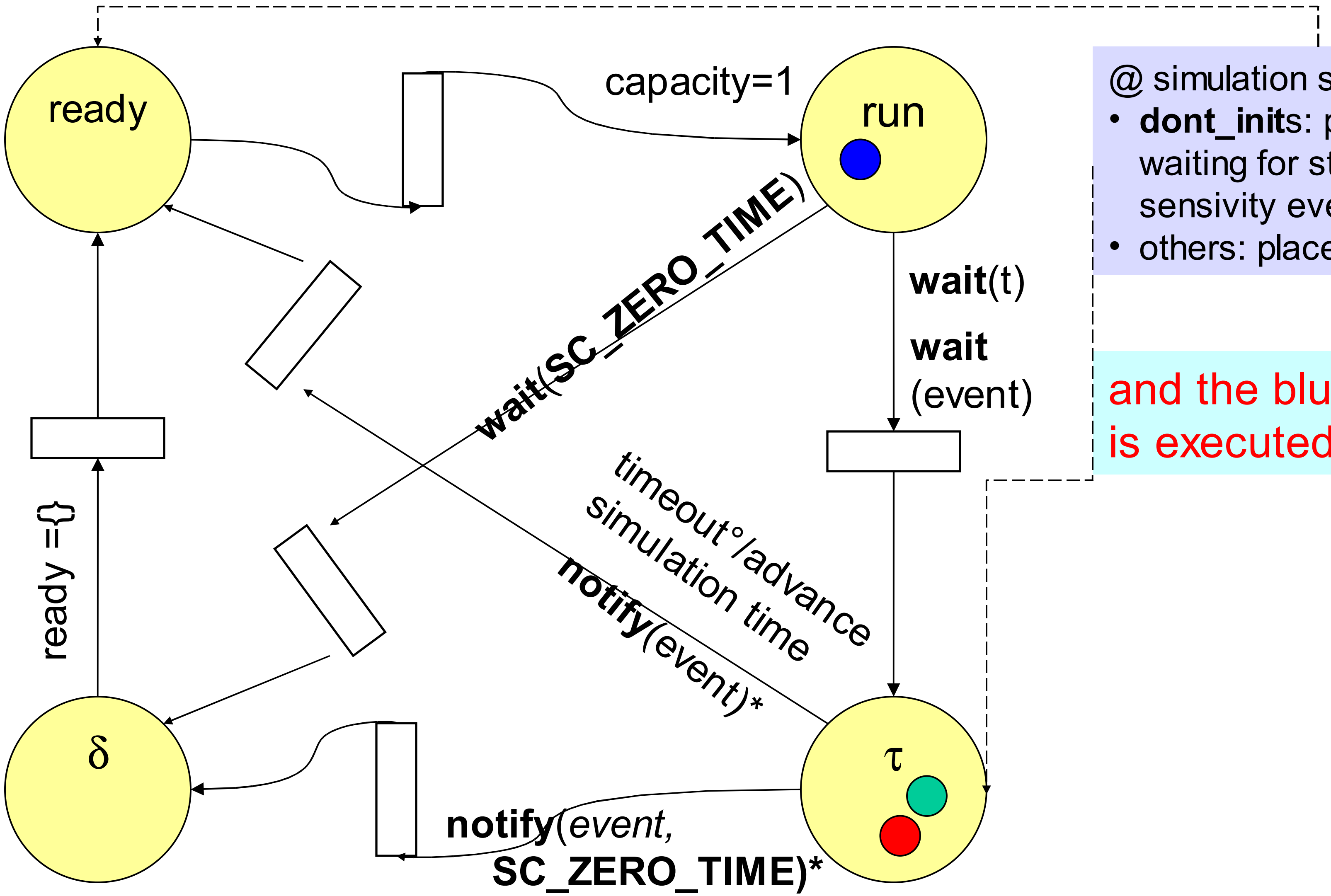
@ simulation start:

- **dont_inits**: placed in τ waiting for static sensivity event
- others: placed in ready

now the blue thread is the 1st thread to become ready; simulation time is advanced.

# Transitions between thread states

## - Predicate/transition net model ≈ activity chart → queuing model -



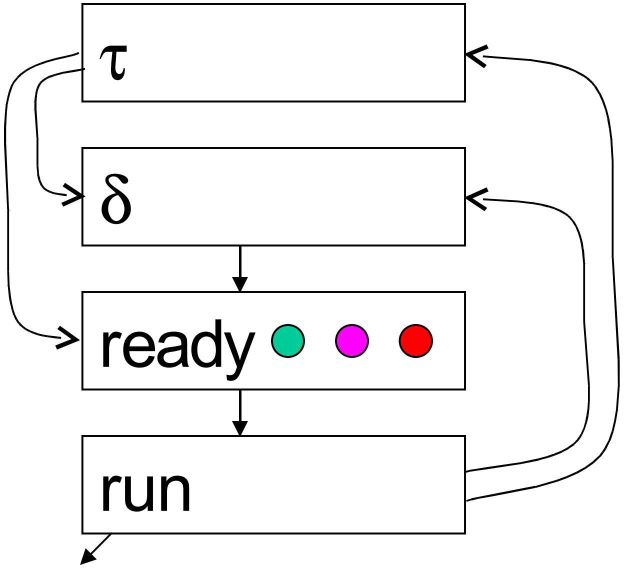
@ simulation start:

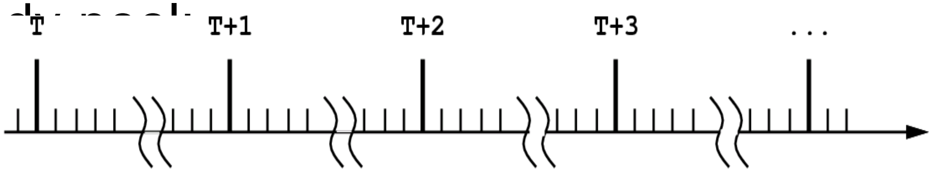
- **dont_init**s: placed in τ waiting for static sensivity event
- others: placed in ready

and the blue thread is executed

# Control flow representation

```
While there is a process in the τ pool                 // macroscopic time loop
  While there is a process in the “δ” pool             // δ time loop
    While there is a process in the ready pool         // same δ time
      {take any process
      execute: evaluate signal changes;
      may send event notifications
        • immediate (→ put into ready pool),
        • delayed (→ “δ” pool, with 0 time)
        • timed (→ waiting pool, with time)
      until process completes (via return) or
      suspends (via calls to wait()); process to
      pool}
    Update pending signal changes; (see
    sc_signal)
    processes waiting in “δ” pool → ready pool
  If ∃ process ∈ τ pool: advance time
```





# Event

Events can have only one pending notification, and retain no „memory“ of past notifications.

Multiple notifications to the same event, without an intermediate trigger are resolved according to the following rule:

An earlier notification will always override one scheduled to occur later, and an immediate notification is always earlier than any δ-cycle delayed or timed notification.

# sc_event_queue

Event queues allow single events to be scheduled multiple times. If events are scheduled for the same time, they will be separated by a  cycle. All scheduled events can be cancelled with the **.cancel_all**() method.

**Example:**

**sc_event_queue** action;
action.**notify**(20,**SC_MS**); // schedule for 20 ms from now
action.**notify**(1.5,**SC_NS**); // schedule for 1.5 ns from now
action.**notify**(**SC_ZERO_TIME**); // for next δ cycle
action.**cancel_all**();

# Summary

- Modules are key design entities and can be created with **SC_MODULE.**
- Constructor **SC_CTOR** or **SC_HAS_PROCESS** required.
- There are two types of processes within modules:
  - **SC_THREAD:** started once, may contain **wait,**
  - **SC_METHOD:** run to completion, cannot call **wait**, may call **next_trigger**.
- 6 different ways of describing structural hierarchy
- Events can be used for notifications
- The simulation cycle uses an evaluation/update cycle for signal updates, corresponding to the δ-cycle of VHDL.