

Synthese Eingebetteter Systeme

Sommersemester 2011

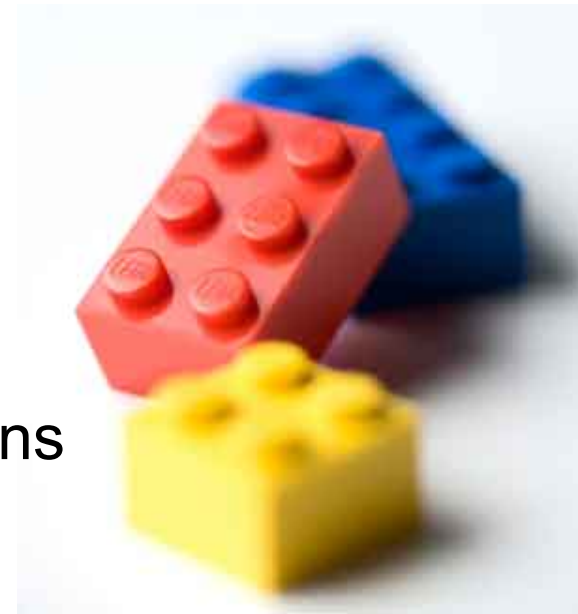
3 – SystemC-Komponenten

Michael Engel
Informatik 12
TU Dortmund

2011/04/11

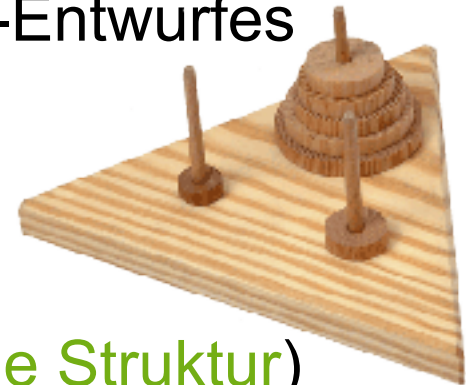
SystemC-Komponenten

- Zeitmodell
- Datentypen
- Instanziierung
- Stolpersteine
- **Module, Hierarchie** und Struktur
- Module und Nebenläufigkeit
- Ereignisse, Sensitivität und Notifications
- Ports, Interfaces und Kanäle
- Kommunikation



Module

- Module sind die Bausteine eines SystemC-Entwurfes
- Ein Modul enthält
 - Prozesse (→ **Funktionalität**)
 - und/oder Untermodule (→ **hierarchische Struktur**)
- Module können
 - mit dem **SC_MODULE**-Makro definiert werden
 - oder als C++-Klasse (→ Übung)



Komponenten der Modulbeschreibung

```
SC_MODULE( module_name ) {  
    // Declaration of module ports  
    // Member channel instances  
    // Member data instances  
    // Member module instances (sub-designs)  
    // Konstruktor  
    // Destruktor  
    // Prozess-Memberfunktionen (Prozesse)  
    // Hilfsfunktionen  
};
```

Der **SC_MODULE** Klassenkonstruktor: **SC_CTOR**

Ein Modul muss genau einen C++-Konstruktor enthalten. Konstruktoren sollten mit dem SystemC-Makro **SC_CTOR** erstellt werden oder **SC_HAS_PROCESS** (*module_name*); verwenden

Syntax von SC_CTOR:

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

Der Konstruktor muss

- Teilentwürfe initialisieren/allozieren
- Teilentwürfe verbinden
- Ereignis-Sensitivitäten definieren
- Prozesse beim SystemC-Kern anmelden

SystemC-Beispiel: Hello, World!

```
#include <Hello_SC.h>

int sc_main(int argc, char* argv[]) {
    const sc_time t_PERIOD(8,SC_NS);
    sc_clock clk("clk",t_PERIOD);
    Hello_SC iHello_SystemC
        ("iHello_SystemC");
    iHello_SystemC.clk_pi(clk);
    sc_start(10);
    return 0;
}
```

Main.cpp

```
#include <Hello_SC.h>

void Hello_SC::main_method(void)
{
    std::cout << sc_time_stamp() <<
        "Hello world!" << std::endl;
}
```

Hello_SC.cpp

```
#ifndef HELLO_SYSTEMC_H
#define HELLO_SYSTEMC_H

#include <systemc.h>
#include <iostream>

SC_MODULE(Hello_SC) {
    sc_in_clk clk_pi;
    void main_method(void);

    SC_CTOR(Hello_SC) {
        SC_METHOD(main_method);
        sensitive << clk_pi.neg();
        dont_initialize();
    }
};
#endif
```

Hello_SC.h

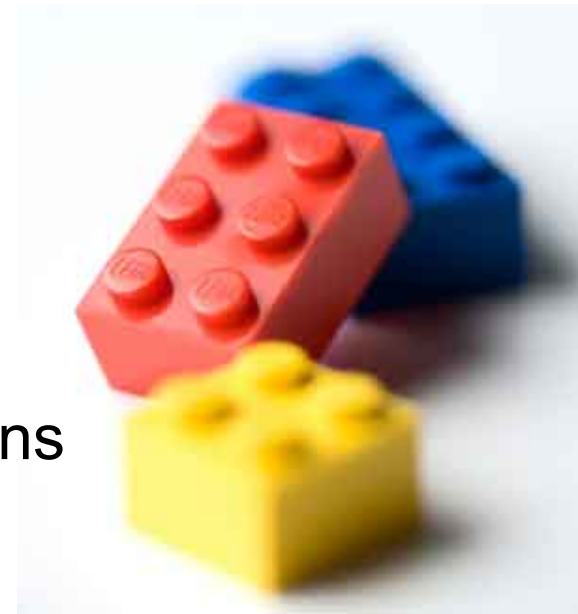
Modul-Definition

Konstruktor-Definition

Ereignis-Sensitivitäten

SystemC-Komponenten

- Zeitmodell
- Datentypen
- Instanziierung
- Stolpersteine
- Module, Hierarchie und Struktur
- Nebenläufigkeit
- Ereignisse, Sensitivität und Notifications
- Ports, Interfaces und Kanäle
- Kommunikation

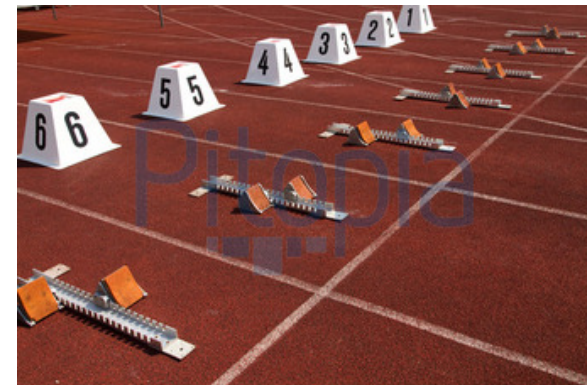


Prozesse

- **Definition:** Ein System-C **process** ist eine *member function* oder Klassenmethode eines **SC_MODULE**.
 - Sie wird vom Scheduler des SystemC-Simulationskerns aufgerufen

Syntax:

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



SC_THREAD

- Threads sind einfache Prozesse
 - Erzeugung durch das SC_THREAD-Makro

- **Syntax:**

```
SC_THREAD(process_name);
```

- Registrierung durch Aufruf des Makros im Konstruktur

- **Kontext:**

```
SC_MODULE(simple_process){  
    SC_CTOR(simple_process) {  
        SC_THREAD(thread_process);  
    }  
    void thread_process(void);  
};
```

SC_THREAD

Mit **SC_THREAD** erzeugte Prozesse

- Werden genau *einmal* gestartet,
- Werden ausgeführt bis zum Aufruf von **wait()** oder **return**,
 - **wait()** kann indirekt durch blockierende Lese-/Schreiboperationen aufgerufen werden
- Enthalten meist Endlosschleifen mit **wait**-Aufrufen,
- Besitzen **Persistenz** für alle lokalen Variablen,
- Entsprechen grob Prozessen in einem Betriebssystem, das kooperatives Multitasking verwendet oder Coroutinen

SC_THREAD::wait()

Beispiele:

- `wait(time);`
- `wait(event);`
- `wait(event [, | event]);`
// fortfahren, wenn (mind.) einer der Events eingetreten ist
- `wait(event [, & event]);`
// Erst fortfahren, wenn **alle** Events eingetreten sind
- `wait(timeout, event);` // Event mit Timeout
- `wait();` // verwendet statische Sensitivität

`time_out()` kann direkt nach `wait` aufgerufen werden, um festzustellen, ob `wait` von einem Timeout beendet wurde (anstelle eines Events)



SC_METHOD

Das **SC_METHOD**–Makro kann zur Erzeugung eines Prozesses innerhalb eines Konstruktors verwendet werden

Ein solcher Prozess

- *Läuft stets bis zu seinem Ende*;
- Darf weder **wait()** noch blockierende **read/write**-Methoden verwenden
- Ist effizienter als mit **SC_THREAD** erzeugte Prozesse
- Verwirft die Werte aller Variablen am Ende



next_trigger()

- Die **next_trigger()**-Methode kann verwendet werden, um die Sensitivität von **SC_METHODs** anzugeben. Der letzte vor dem Ende einer **SC_METHOD** stehende Aufruf von **next_trigger** gibt die Bedingung an, die für die nächste Aktivierung der Methode gültig ist.

Beispiele:

- `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 sensitivity`

Statische Sensitivität

- **Statische Sensitivität** ist eine Alternative zur Verwendung von **wait()** oder **next_trigger()**.
 - Wird zur Übersetzungszeit festgelegt
- Statische Sensitivität wird bestimmt durch die aktuellste Prozessregistrierung
- Aufruf:
 - **sensitive** << *event* [<<*event*] ...;
 - Ältere Schreibweise **sensitive** (*event* [, *event*] ..); ist *deprecated*

don't_initialize

- Normalerweise werden alle Prozesse bei Simulationsstart initialisiert
- Soll ein Prozess *nicht* initialisiert werden, muss **dont_initialize()** verwendet werden
- Der Aufruf muss der Prozessregistrierung direkt folgen
- Prozesse mit **dont_initialize**, sollten eine statische Sensitivitätsliste verwenden
 - Sonst wird der Prozess nie ausgeführt

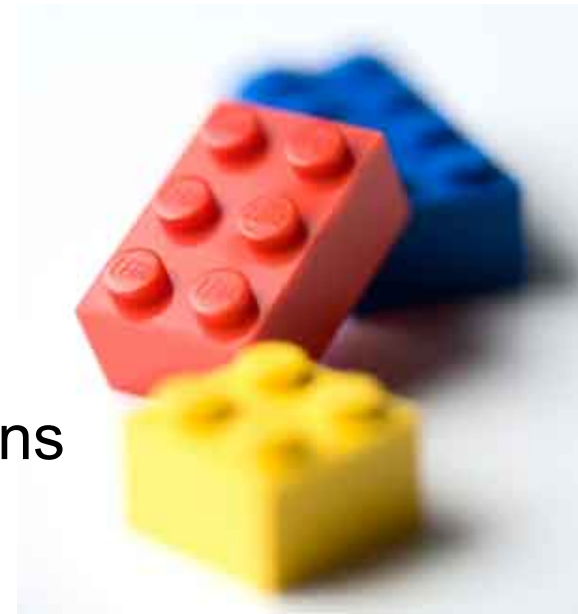
Beispiel:

```
SC_METHOD(attendand_method);  
  sensitive(fillup_request);  
  dont_initialize();
```

...

SystemC-Komponenten

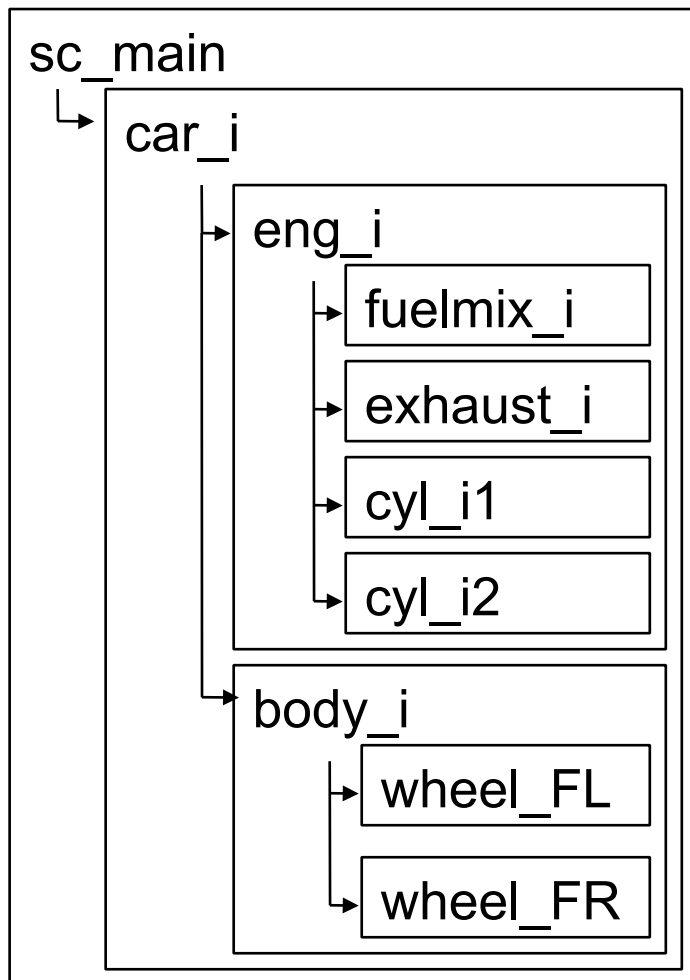
- Zeitmodell
- Datentypen
- Instanziierung
- Stolpersteine
- Module, Hierarchie und **Struktur**
- Nebenläufigkeit
- Ereignisse, Sensitivität und Notifications
- Ports, Interfaces und Kanäle
- Kommunikation



Entwurfshierarchie



Beispiel



Unterschiedliche Ansätze zur Instanziierung von Modulen:

- in .h oder .cc-Datei,
- Direkte Instanziierung in der Deklaration oder dynamische Erzeugung mit Pointern,
- Auf oberster Ebene der Hierarchie oder auf einer tieferliegenden Ebene

...und Kombinationen.



Direkte Instanziierung auf oberster Ebene

Beispiel:

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

Einfach zu realisieren

Indirekte Instanziierung auf oberster Ebene

Beispiel:

```
// Datei: main.cpp
#include "Wheel.h"
```

```
int sc_main (int argc, char* argv[] {
    Wheel* wheel_FL; // Pointer auf Rad vorne links
    Wheel* wheel_FR; // Pointer auf Rad vorne rechts
    wheel_FL = new Wheel("wheel_FL"); // erzeuge FL
    wheel_FR = new Wheel("wheel_FR"); // erzeuge FR
    sc_start();
    delete wheel_FL;
    delete wheel_FR;
}
```

Mehr Code, aber auch flexibler:
Schleifen, Arrays,
Fallunterscheidungen möglich

Direkte Instanziierung von Untermodulen im Header

Beispiel:

```
// Datei: Body.h
#include "Wheel.h"
SC_MODULE(Body) {
    Wheel wheel_FL;
    Wheel wheel_FR;
    SC_CTOR(Body)
        : wheel_FL("wheel_FL"), // Initialisierung
          wheel_FR("wheel_FR") // Initialisierung
    {
        // andere Initialisierungen
    }
}
```

Initializer list to be used
due to C++ rules

Indirekte Instanziierung von Untermodulen im Header

Beispiel:

```
// Datei: 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");
```

```
        // andere Initialisierungen
```

```
    }
```

```
};
```

Einfacher zu lesen als
die direkte Methode

Direkte Instanziierung von Untermodulen

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

Verschiebt
Initialisierungs-Aufwand
von Header-Datei zur
Implementierung hin,
für Nutzer des Headers
transparent

```
// Datei: Body.cpp
#include "Body.h" // Konstruktor:
Body::Body(sc_module_name nm)
: wheel_FL("wheel_FL"),
  wheel_FR("wheel_FR"),
  sc_module(nm)
{ .. andere Initialisierungen..}
```

Indirekte Instanziierung von Untermodulen

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

Verschiebt
Initialisierungs-Aufwand
von Header-Datei zur
Implementierung hin,
für Nutzer des Headers
transparent

```
// Datei: Body.cpp
#include "Wheel.h" // Konstruktor:
Body::Body(sc_module_name nm)
: sc_module(nm)
{wheel_FL=new Wheel("wheel_FL");
 wheel_FR=new Wheel("wheel_FR");
 .. andere Initialisierungen.. }
```

Implementierungsansätze

| Ebene | Allokation | Vorteile | Nachteile |
|---------|------------------------------------|---|----------------------------------|
| Oberste | Direkt | Minimaler Code | Inkonsistent mit anderen Ebenen |
| Oberste | Indirekt | Dynamisch konfigurierbar | Pointer benötigt |
| Modul | Direkt Nur Header | Alles in einer Datei, leicht zu verstehen | Erfordert Header für Untermodule |
| Modul | Indirekt Nur Header | Alles in einer Datei, dynamisch konfigurierbar | Pointer benötigt |
| Modul | Direkt mit separater Übersetzung | Verbirgt Implementierung | Erfordert Header für Untermodule |
| Modul | Indirekt mit separater Übersetzung | Verbirgt Untermodul-Header und -implementierung, dynamisch konfigurierbar | |

SystemC-Beispiel: Hello, World!

```
#include <Hello_SC.h>

int sc_main(int argc, char* argv[]) {
    const sc_time t_PERIOD(8,SC_NS);
    sc_clock clk("clk",t_PERIOD);
    Hello_SC iHello_SystemC
        ("iHello_SystemC");
    iHello_SystemC.clk_pi(clk);
    sc_start(10);
    return 0;
}
```

Main.cpp

```
#include <Hello_SC.h>

void Hello_SC::main_method(void)
{
    std::cout << sc_time_stamp() <<
    "Hello world!" << std::endl;
}
```

Hello_SC.cpp

```
#ifndef HELLO_SYSTEMC_H
#define HELLO_SYSTEMC_H

#include <systemc.h>
#include <iostream>

SC_MODULE(Hello_SC) {
    sc_in_clk clk_pi;
    void main_method(void);

    SC_CTOR(Hello_SC) {
        SC_METHOD(main_method);
        sensitive << clk_pi.neg();
        dont_initialize();
    }
};
#endif
```

Hello_SC.h

Modul-Definition

Konstruktor-Definition

Ereignis-Sensitivitäten

SystemC-Komponenten

- Zeitmodell
- Datentypen
- Instanziierung
- Stolpersteine
- Module, Hierarchie und Struktur
- Nebenläufigkeit
- Ereignisse, Sensitivität und Notifications
- Ports, Interfaces und Kanäle
- Kommunikation



Events

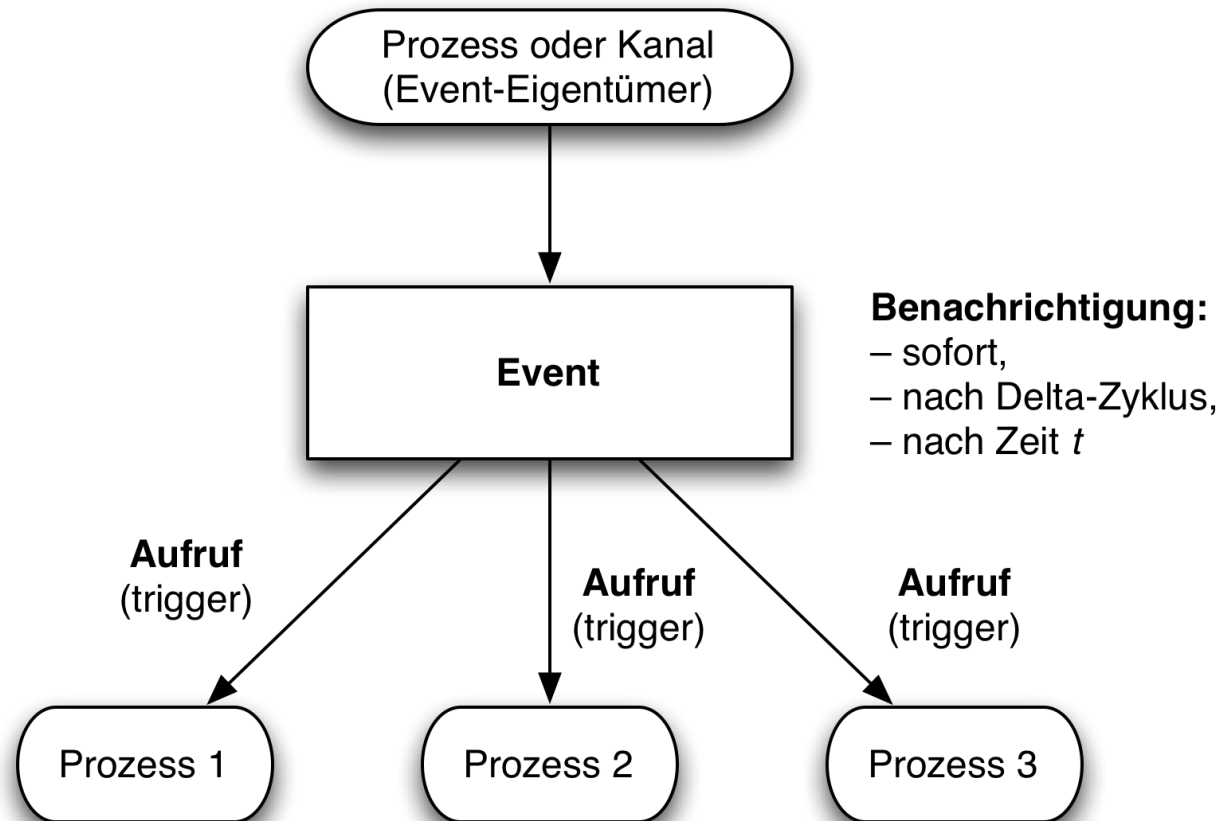
- Ein Ereignis (Event) ist ein Objekt, das durch **sc_events**, dargestellt wird
 - Entscheidet, ob und wann die Ausführung eines Prozesses gestartet oder fortgesetzt werden soll
- Events stellen Bedingungen dar, die während des Simulationslaufs eintreten können und die entsprechend einen Prozess aktivieren sollen
- Die Klasse **sc_event** stellt Synchronisationsprimitive für Prozesse zur Verfügung

Events

- Unterscheidung zwischen
 - Event (“Möglichkeitsform“) und
 - Eigentlichem *Auftreten* eines Events (“anzeigende Form“)
- Ein Event ist normalerweise mit einer Zustandsänderung in einem Prozess oder eines Kanals verbunden
- Der *Besitzer* des Events hat dem Event-Objekt Änderungen anzuzeigen
- Dieser Vorgang des Anzeigens der Event-Änderung wird **Benachrichtigung (notification)** genannt

Event

Event-Benachrichtigung und Prozessaufruf



Auslösen von Events: .notify

- Events werden mit **.notify()** ausgelöst
 - **Sofortige notify**s verschieben wartende Prozesse sofort von “wartend” nach “ready”

Examples:

```
event_name.notify(); notify(event_name);
```

- **Verzögerte Benachrichtigung:**

```
event_name.notify(SC_ZERO_TIME);  
notify(event_name, SC_ZERO_TIME);
```

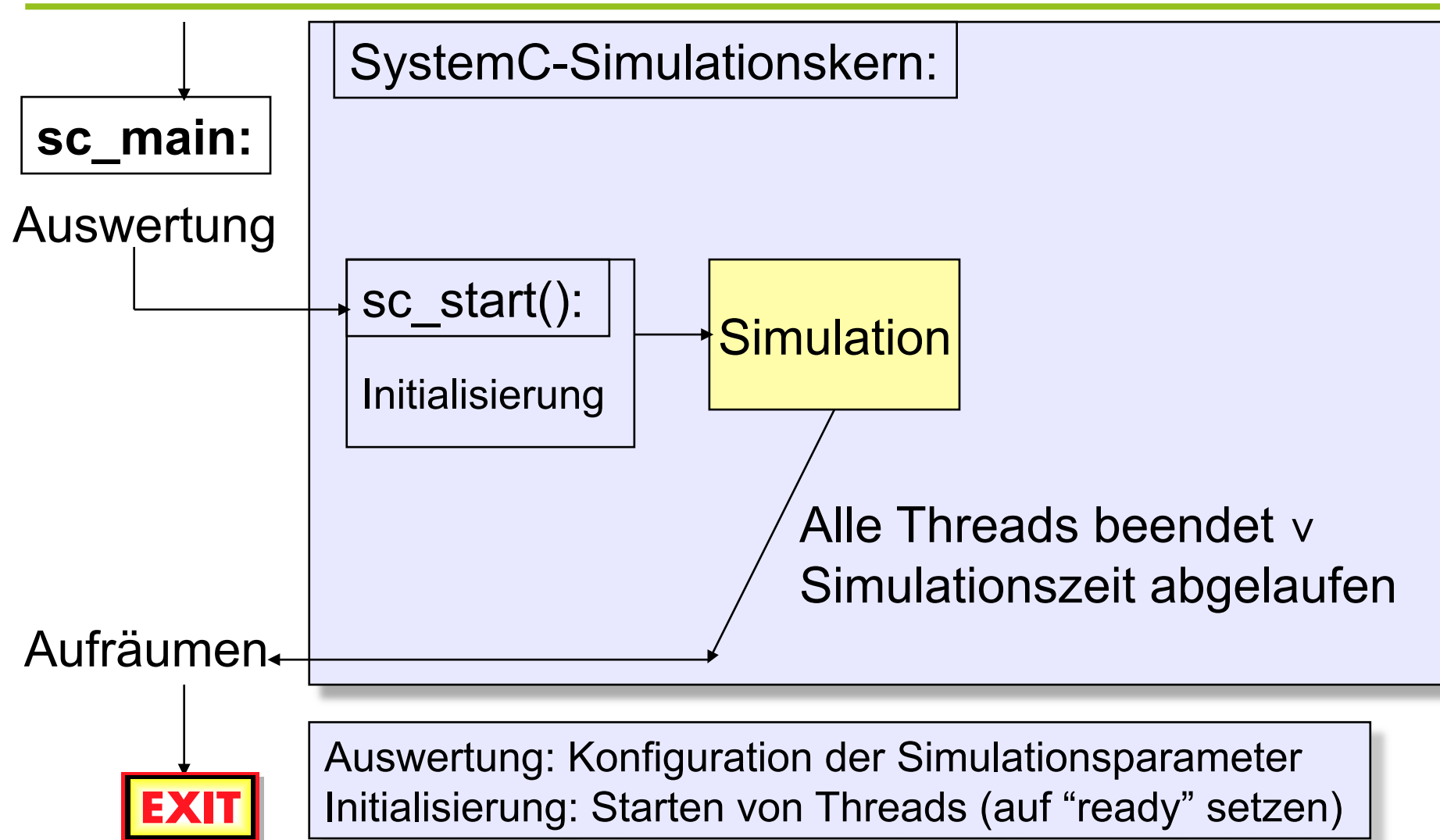
Wartende Prozesse werden beim folgenden δ -Zyklus ausgeführt

- **Zeitabhängige Benachrichtigung** führt zu Prozessausführung nach angegebener Zeit

```
event_name.notify(t); notify(event_name, t);
```

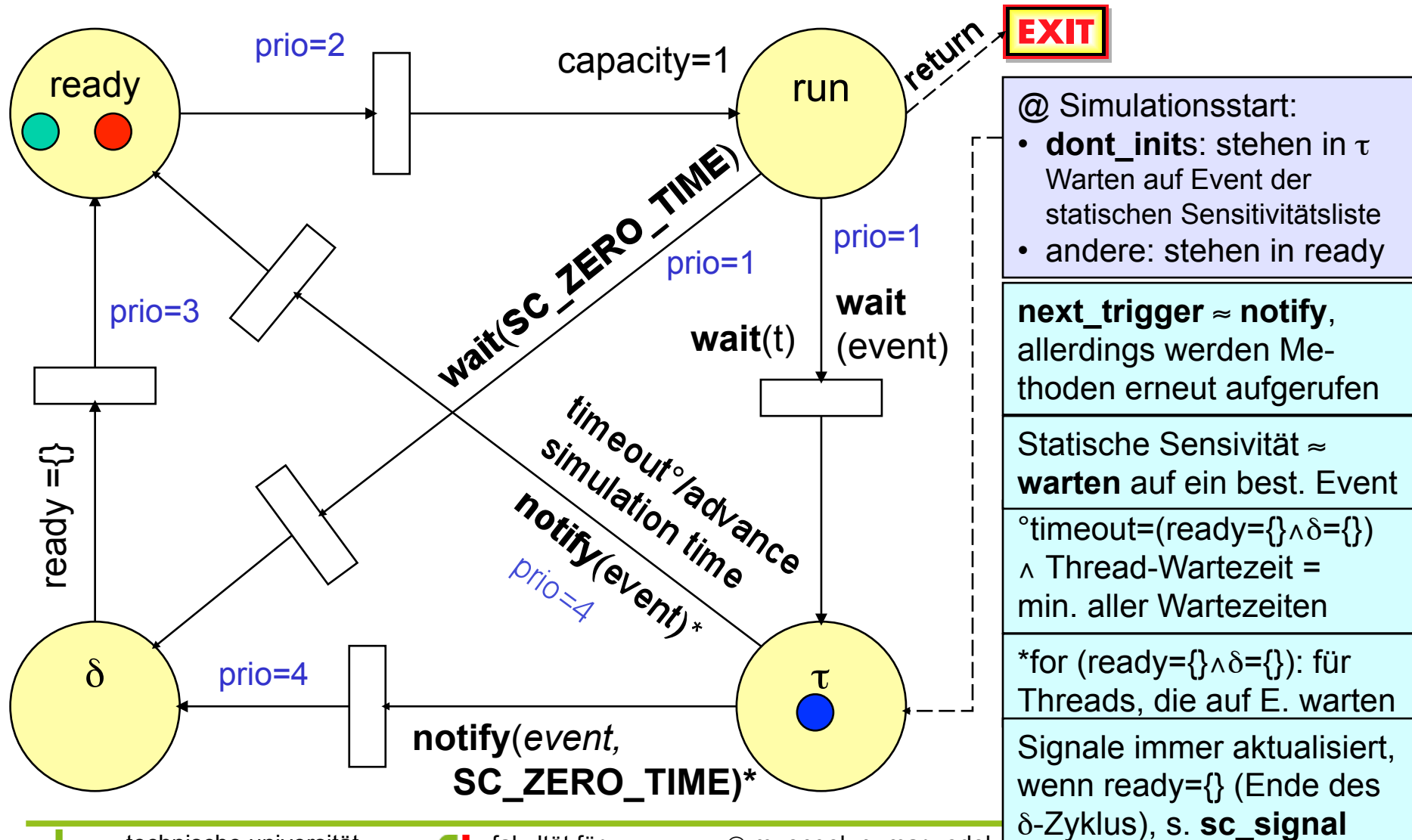


Simulations-System (vereinfacht)



Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



@ Simulationsstart:

- **dont_inits**: stehen in τ Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

next_trigger \approx notify, allerdings werden Methoden erneut aufgerufen

Statische Sensivität \approx **warten** auf ein best. Event

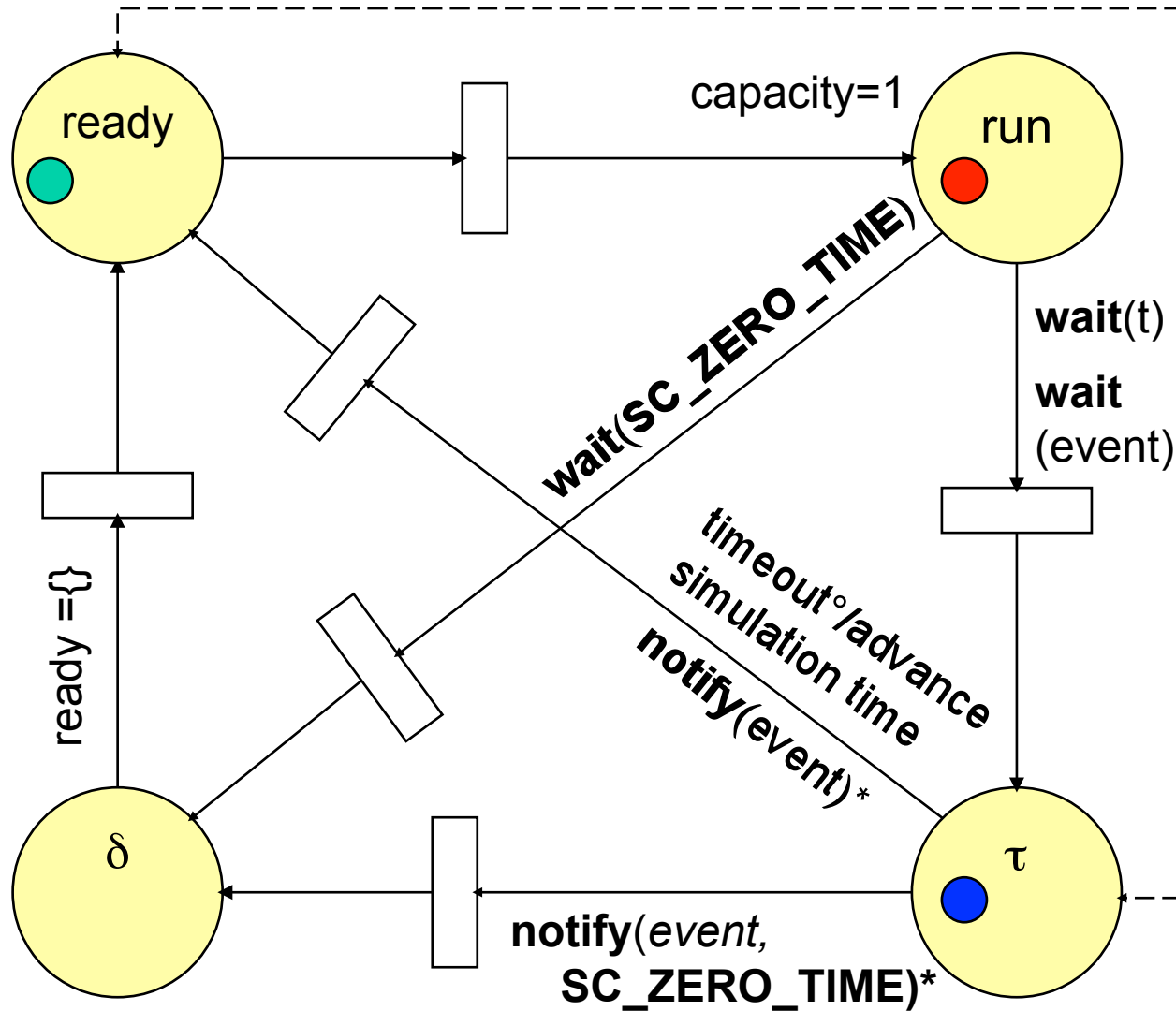
$^\circ$ timeout=(ready={} \wedge δ ={}) \wedge Thread-Wartezeit = min. aller Wartezeiten

*for (ready={} \wedge δ ={}): für Threads, die auf E. warten

Signale immer aktualisiert, wenn ready={} (Ende des δ -Zyklus), s. **sc_signal**

Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



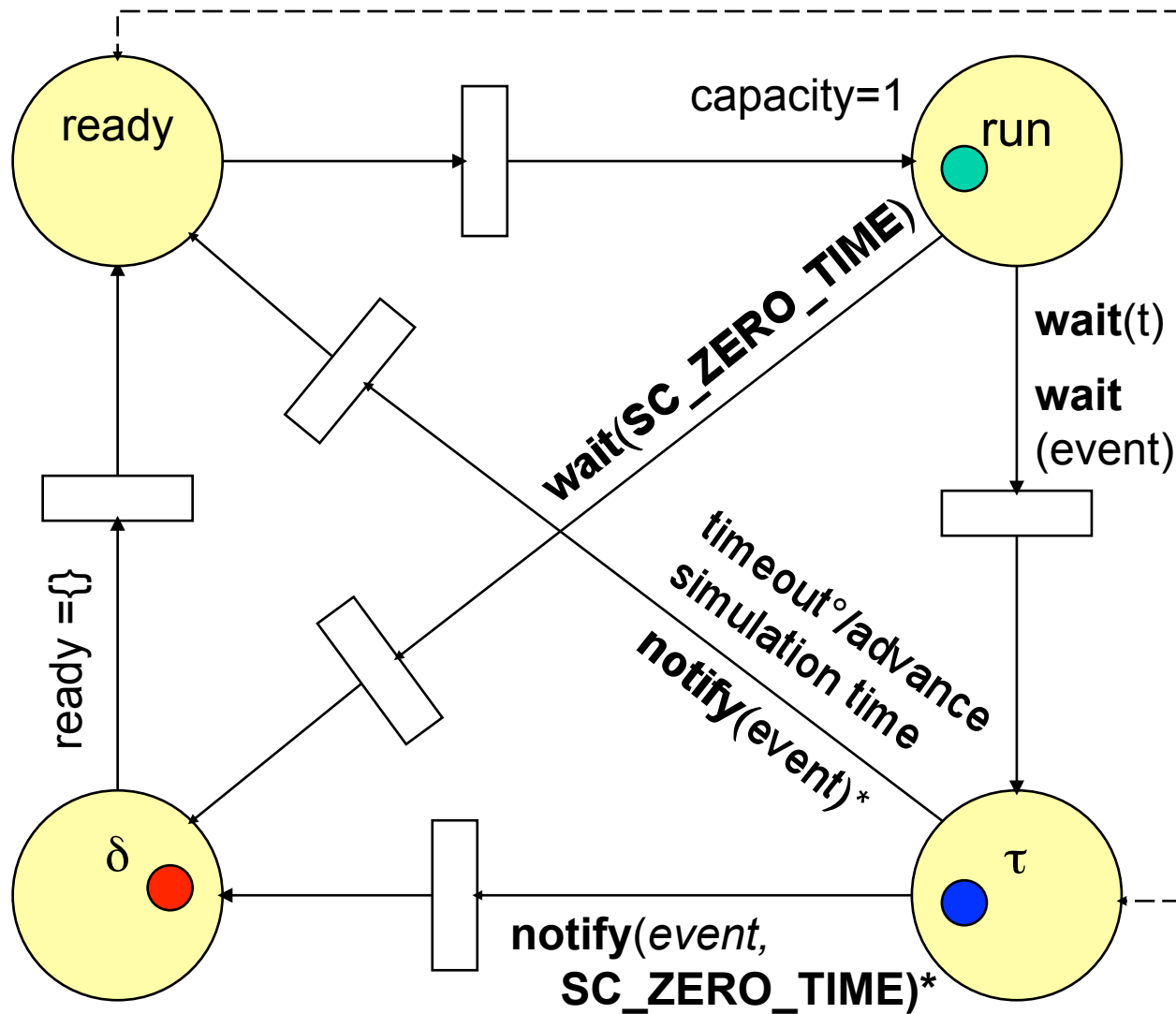
@ Simulationsstart:

- **dont_inits**: stehen in τ
Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

Auswahl zwischen Threads in "ready" *nichtdeterministisch*

Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



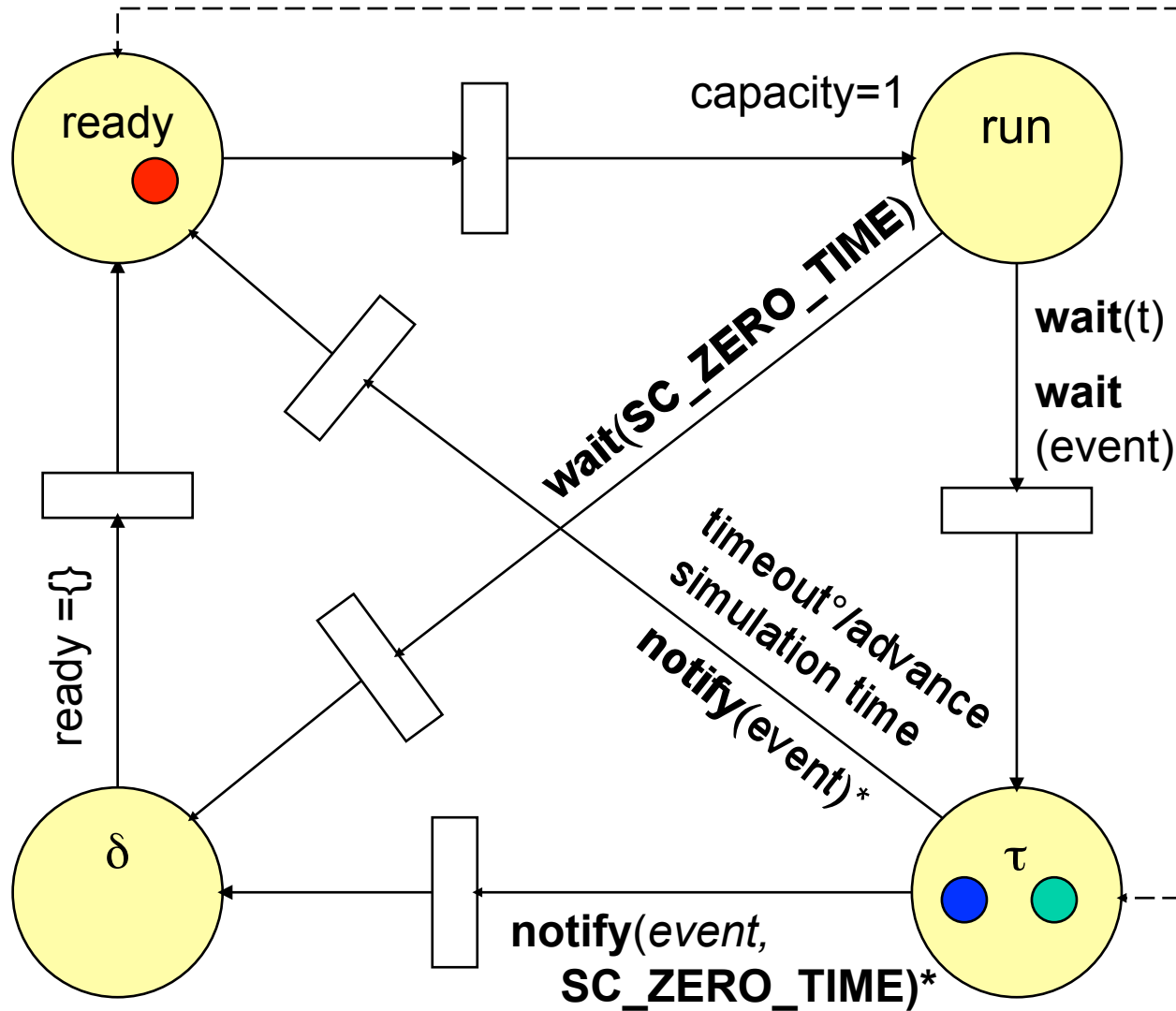
@ Simulationsstart:

- **dont_inits**: stehen in τ Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

“ready”-Zustand leer vor Start des folgenden δ -Zyklus

Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



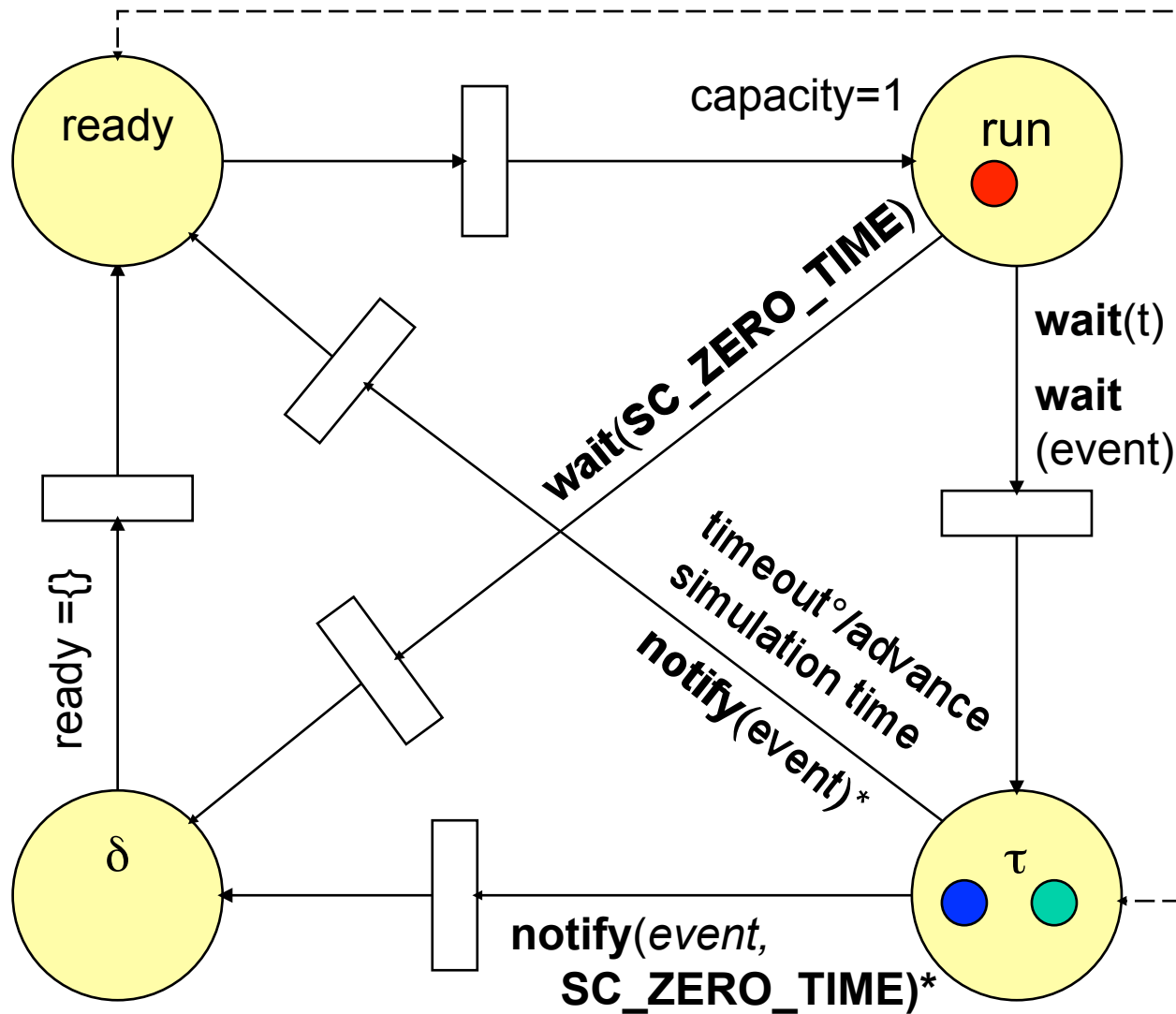
@ Simulationsstart:

- **dont_inits**: stehen in τ Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

Nächster δ -Zyklus

Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



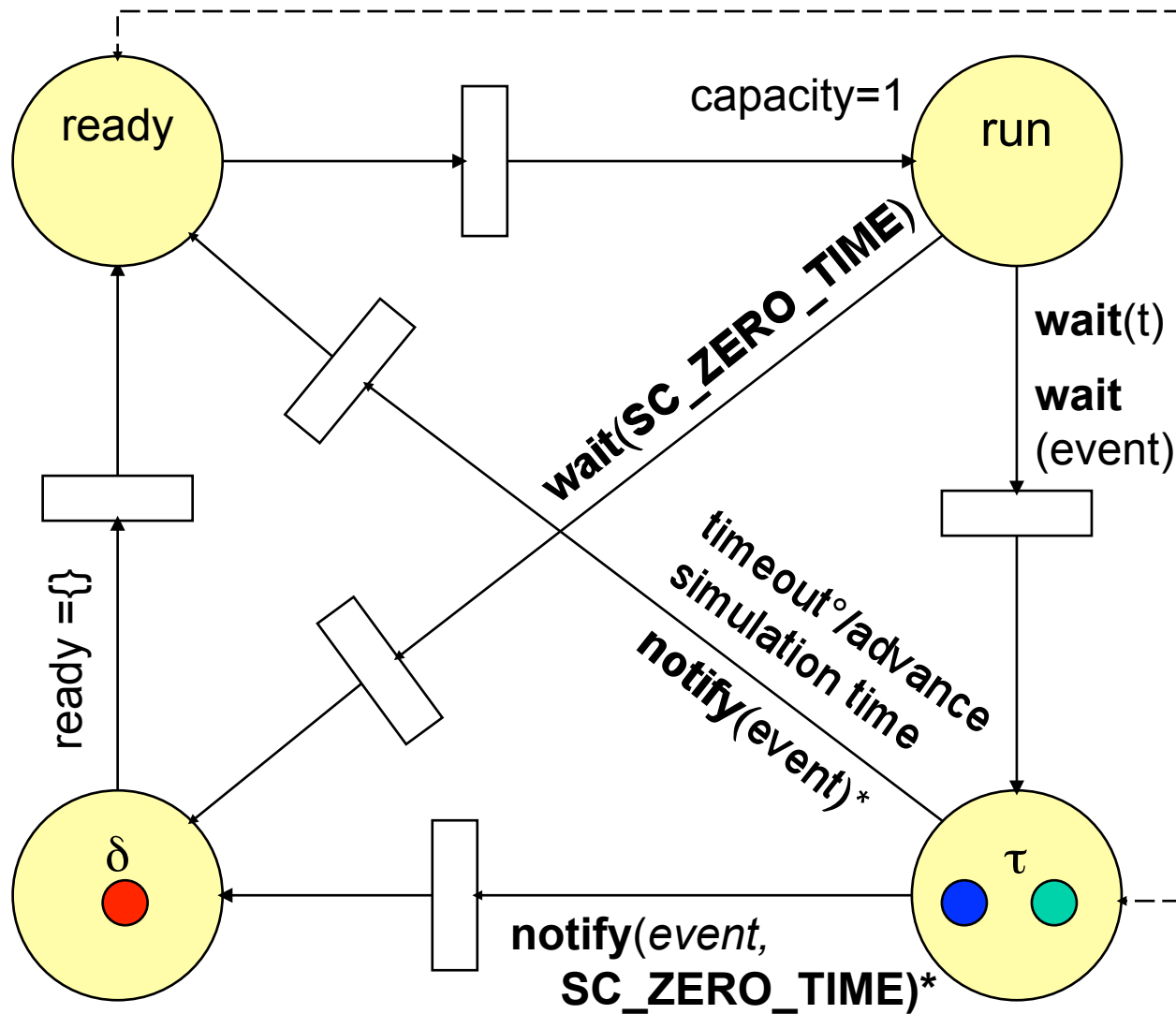
@ Simulationsstart:

- **dont_inits**: stehen in τ Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

Nächster δ -Zyklus ...

Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



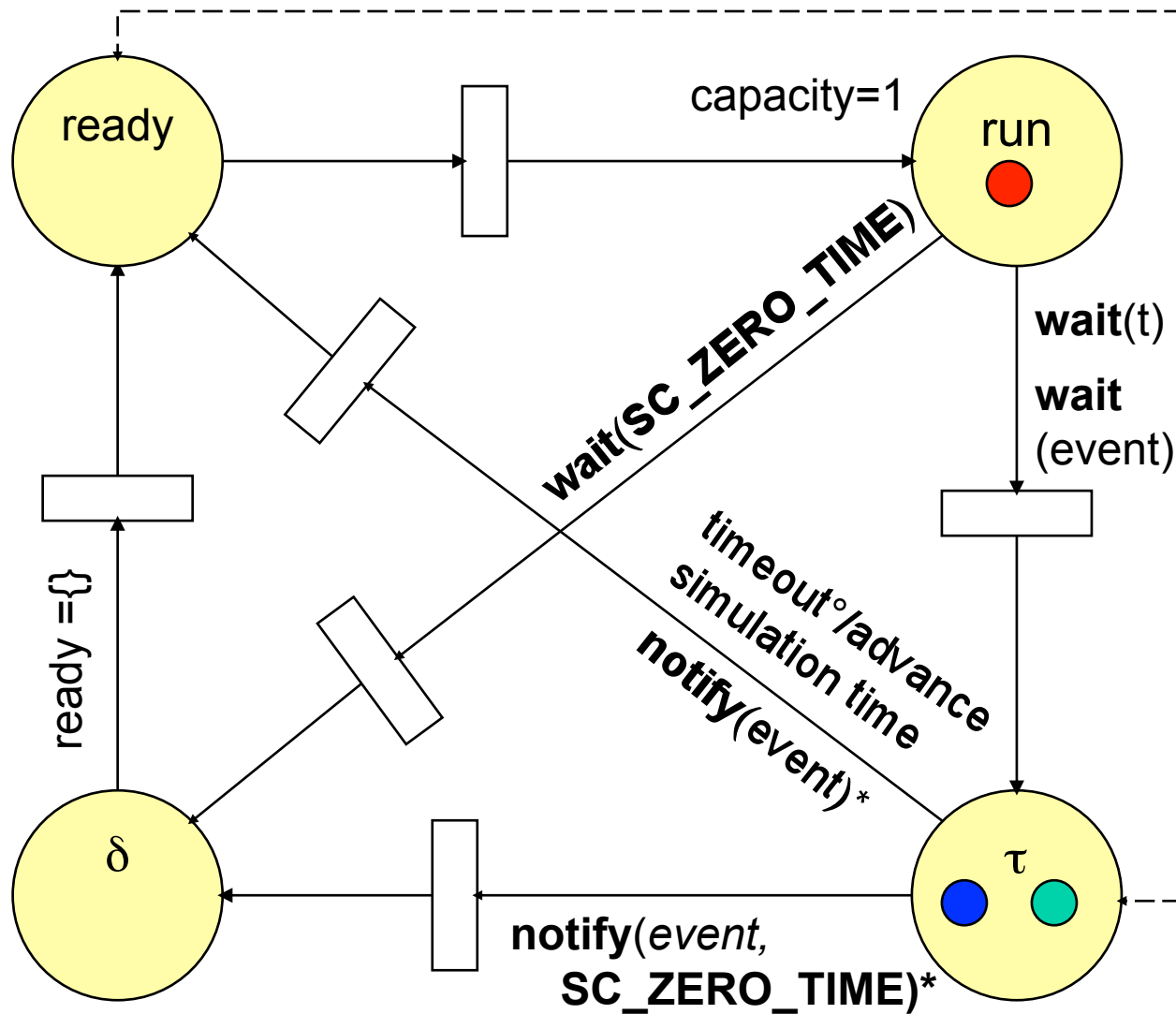
@ Simulationsstart:

- **dont_inits**: stehen in τ Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

Es kann mehrere δ -Zyklen geben

Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



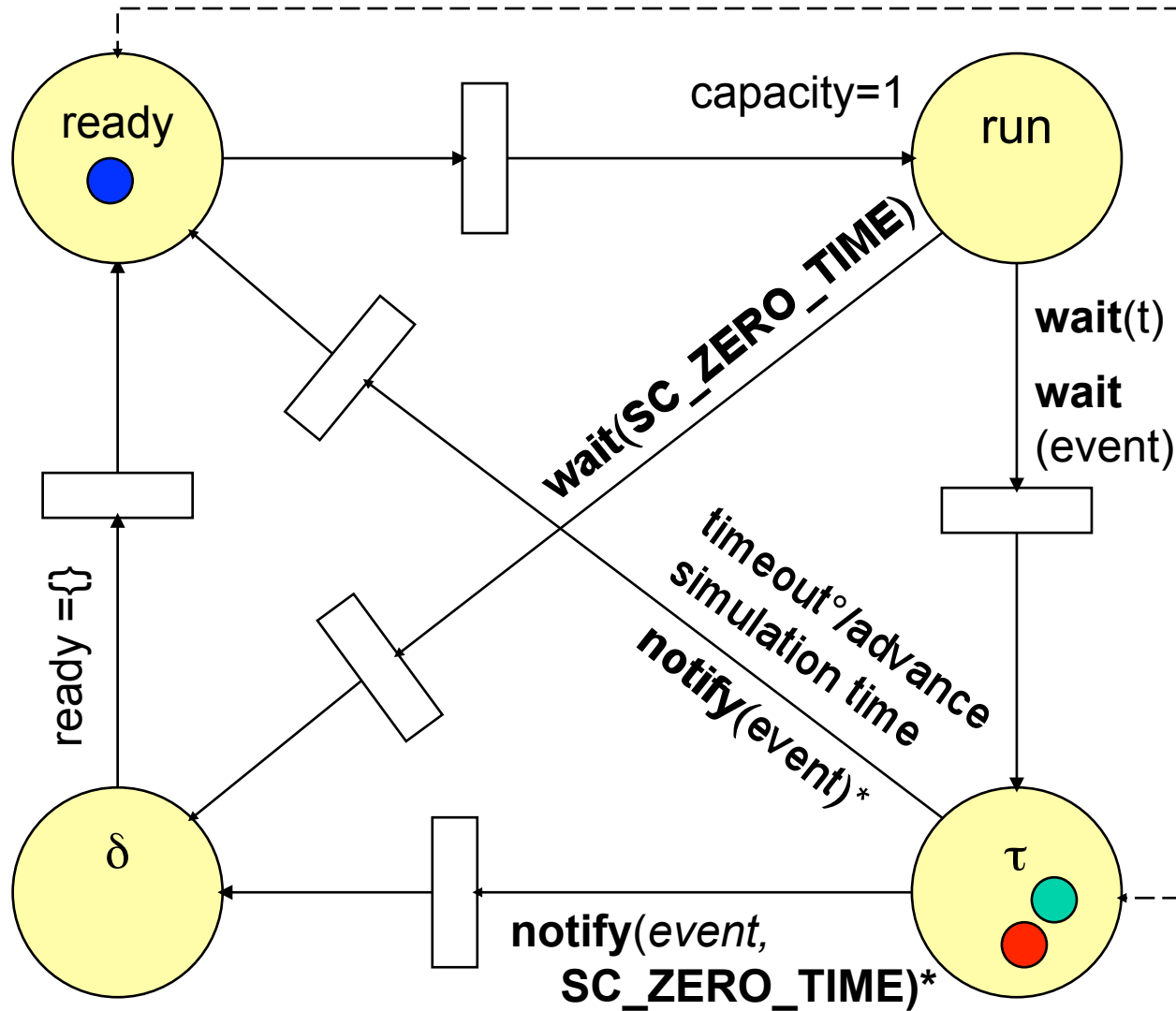
@ Simulationsstart:

- **dont_inits:** stehen in τ Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

Es kann mehrere δ -Zyklen geben...

Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



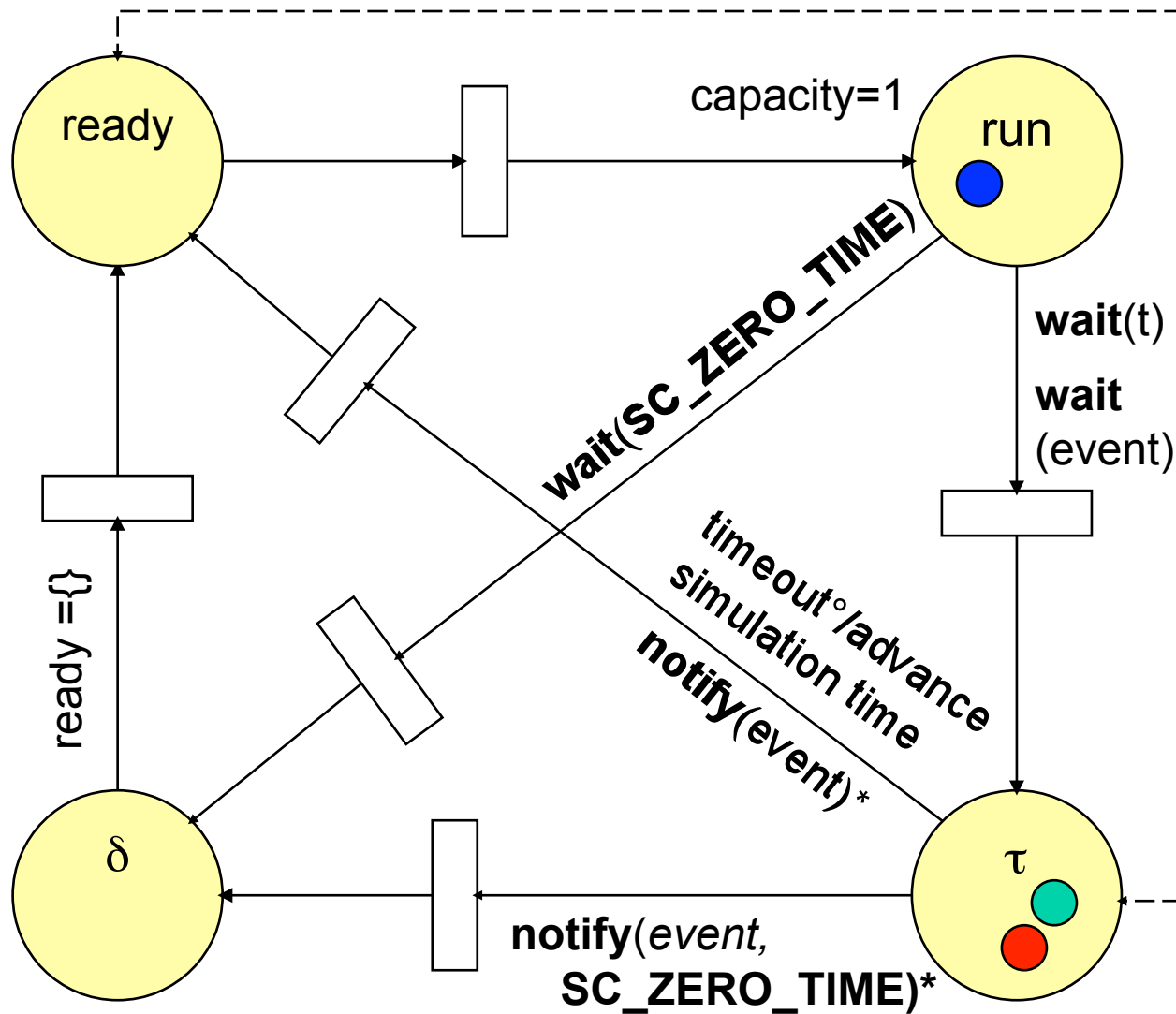
@ Simulationsstart:

- **dont_inits**: stehen in τ Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

Jetzt wird der blaue Thread als Erster "ready". Die Simulationszeit läuft nun weiter

Übergänge zwischen Threadzuständen

– Prädikats-/Transitionsnetz \approx Aktivitäts-Chart \rightarrow Warteschlangenmodell –



@ Simulationsstart:

- **dont_inits**: stehen in τ Warten auf Event der statischen Sensitivitätsliste
- andere: stehen in ready

Und der blaue Thread wird ausgeführt

Darstellung des Kontrollflusses

Solange ≥ 1 Prozess in τ ist

Solange ≥ 1 Prozess in “ δ ”

Solange ≥ 1 Prozess in “ready”

{ Wähle beliebigen Prozess

Ausführung: Evaluierung von Signaländerungen;

Erzeugt evtl. Event-Benachrichtigungen

- Sofort (\rightarrow in “ready” stellen),
- Verzögert (\rightarrow in “ δ ” stellen mit Zeit 0)
- Zeitabhängig (\rightarrow Wartezustand mit Zeit)

Bis Prozess beendet (**return**) oder suspendiert (**wait()**-Aufruf); Prozess an “ready” }

Aktualisiere ausstehende Signaländerungen; (s. **sc_signal**)

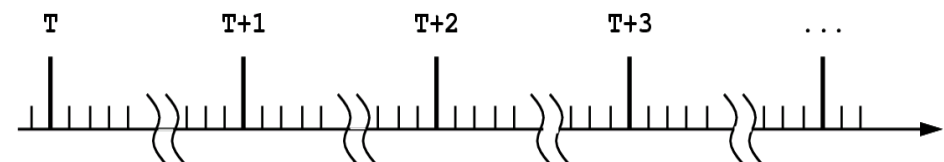
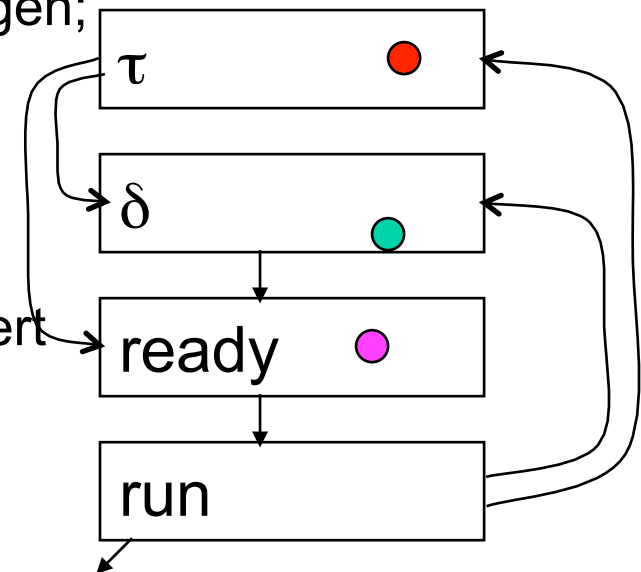
Prozesse in “ δ ” \rightarrow “ready”;

Falls \exists Prozess $\in \tau$: Zeit erhöhen

// Macroscopische Zeit

// δ -Zeit

// selbe δ -Zeit



Events

- Events können nur eine ausstehende Benachrichtigung haben, sie besitzen kein Gedächtnis für vorherige Benachrichtigungen (Verhalten: wie Signale in Unix)
- Mehrfache Benachrichtigungen des selben Events ohne zwischenzeitlichen Prozessaufruf folgen der Regel:

Eine frühere Benachrichtigung überschreibt eine, die später stattfinden soll. Eine sofortige Benachrichtigung finden immer vor allen deklarierten δ -Zyklen oder zeitabhängiger Benachrichtigung statt

sc_event_queue

- Event-Warteschlangen erlauben die mehrfache Erzeugung eines einzelnen Events. Wenn Events zur gleichen Zeit auftreten sollen, werden sie durch einen Zyklus getrennt
- Alle geplanten Events können durch Aufruf der **.cancel_all()**-Methode gelöscht werden

Beispiel:

```
sc_event_queue action;  
action.notify(20,SC_MS); // In 20 ms  
action.notify(1.5,SC_NS); // In 1.5 ns  
action.notify(SC_ZERO_TIME); // in folgendem  $\delta$ -Zyklus  
action.cancel_all();
```

Zusammenfassung

- Module sind die wichtigsten Entwurfseinheiten
 - Erzeugung mit **SC_MODULE**
 - Konstruktor **SC_CTOR** oder **SC_HAS_PROCESS** erforderlich
- Zwei Arten von Prozessen in Modulen:
 - **SC_THREAD**: einmal gestartet, kann **wait** enthalten,
 - **SC_METHOD**: läuft bis Ende, darf **wait** nicht aufrufen, kann **next_trigger** aufrufen
- Strukturelle Hierarchie: 6 verschiedene Beschreibungen
- Events können für Benachrichtigungen verwendet werden
- Der Simulationszyklus verwendet einen Auswerte-/Aktualisierungszyklus für Signalaktualisierungen
 - Entspricht δ -Zyklen von VHDL