

Synthese Eingebetteter Systeme

Wintersemester 2012/13

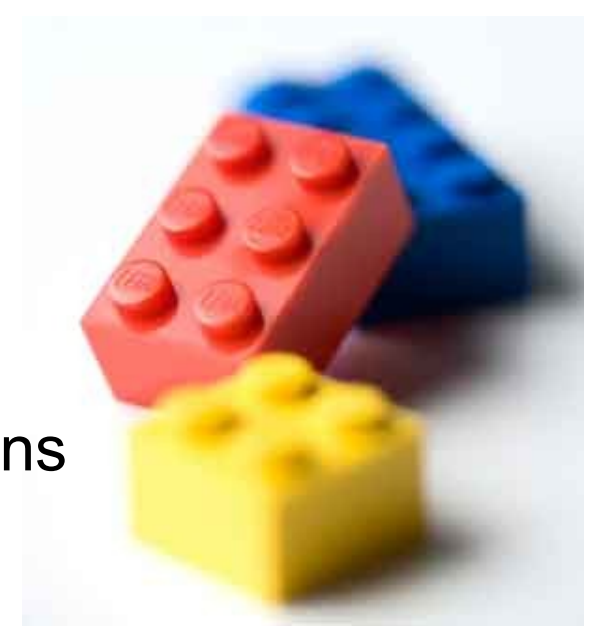
5 – SystemC-Kommunikation

Michael Engel
Informatik 12
TU Dortmund

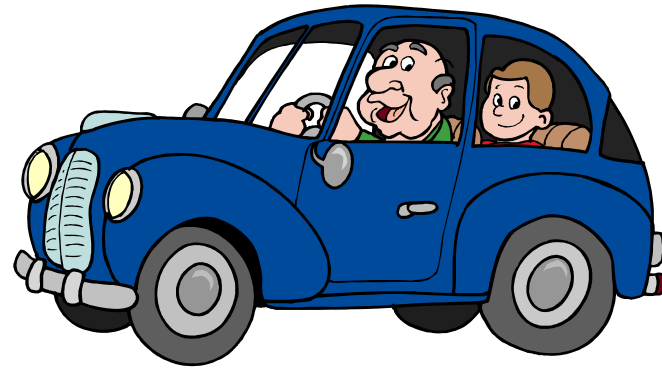
2012/10/31

SystemC-Komponenten

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



Beispiel: Tankstelle (1)



Kunde 1

Volltanken!



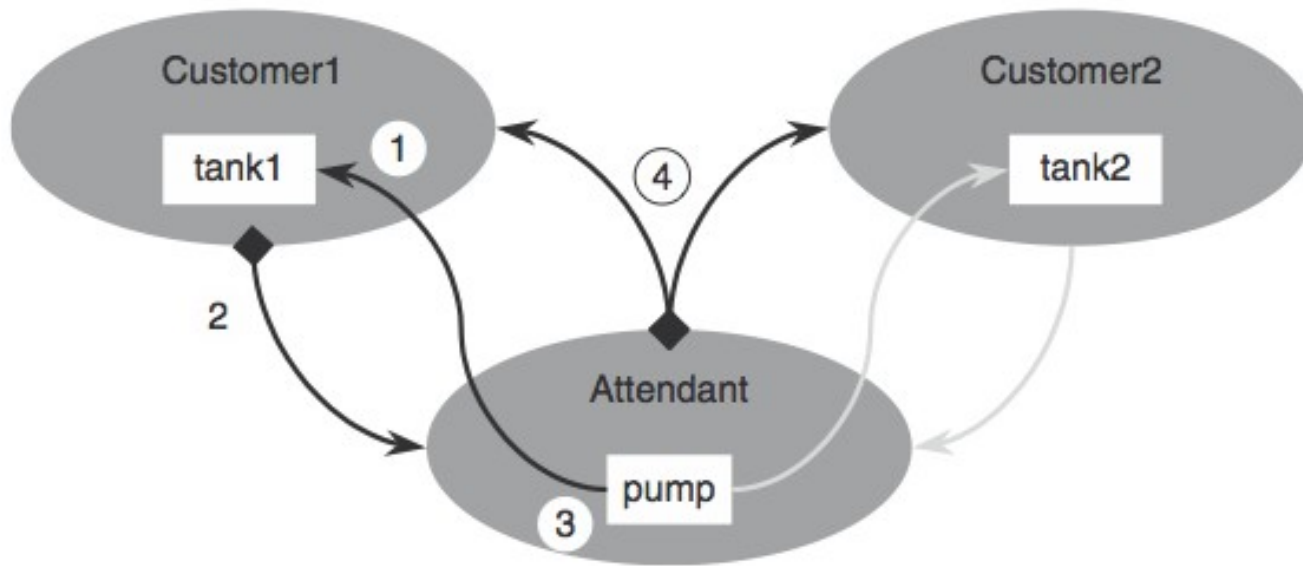
Voll!



Kunde 2

Tankstelle: Ablauf

- Kunde bemerkt leeren Tank, kommt an Tankstelle an (1)
- Tankwart bemerkt, dass Kunde tanken möchte (2)
- ...und muss dies auch erkennen, wenn er gerade dabei ist, einen anderen Kunden zu bedienen (3)



Tankstelle: Ablauf

- Wenn zwei Kunden ankommen, kann der Tankwart auf die Anforderung beider Kunden warten mit
 - **wait (e_request1|e_request2)**
- Semantik von **sc_event** ermöglicht es dem Tankwart nicht zu wissen, *von welchem Kunden* die Anforderung kam
 - Tankwart weiss nicht, welche Anforderung dazu führte, dass der **wait()**-Aufruf zurückkam
- Daher verwendet das Tankstellen-Modell den Status des Tanks als Anzeige, ob der Tank gefüllt werden soll
 - Kunde muss entsprechend überprüfen, ob der Tank tatsächlich gefüllt wurde, wenn der Tankwart dies behauptet (4)

Beispiel: Tankstelle (2)

main.cpp

```
#include <systemc.h>
#include "gas_station.h"
unsigned errors = 0;

char* simulation_name = "gas_station";
int sc_main(int argc, char* argv[]) {
    sc_set_time_resolution(1,SC_NS);
    sc_set_default_time_unit(1,SC_NS);

    gas_station Charlies("Charlies",/*full1*/10,/*full2*/12,/*filltime*/1.5,/*maxfills*/10);

    cout << "INFO: Starting gas_station simulation" << endl;
    sc_start();
    cout << "INFO: Exiting gas_station simulation" << endl;
    cout << "INFO: Simulation " << simulation_name
        << " " << (errors?"FAILED":"PASSED")
        << " with " << errors << " errors" << endl;
    return errors?1:0;}

```

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

```

SC_MODULE(gas_station) { // Lokale Daten
    const sc_time t_MIN; bool m_filling; // state of attendant
    double m_full1, m_full2; double m_filltime; sc_event e_request1, e_request2;
    double m_tank1, m_tank2; unsigned m_count1, m_count2, m_maxcount;
    sc_event e_filled; // Konstruktor
    SC_HAS_PROCESS(gas_station);
    gas_station(sc_module_name _name,
        double full1=10.0, double full2=11.1, double filltime=1.8, unsigned maxcount=5 ):
        sc_module(_name), m_full1(full1), m_full2(full2), m_filltime(filltime), m_tank1(full1),
        m_tank2(full1), m_count1(0), m_count2(0), m_maxcount(maxcount),
        m_filling(false), t_MIN(1,SC_NS)) // 1 Minute Echtzeit = 1 Nanosekunde
    { SC_THREAD(customer1_thread);
        sensitive << e_filled;
        SC_THREAD(Ende customer2_thread);
        SC_METHOD(attendant_method);
        sensitive << e_request1 << e_request2;
        dont_initialize();
    } // Ende Konstruktor (gas_station)
// Prozessdeklarationen
void customer1_thread(void);
void customer2_thread(void);
void attendant_method(void);
std::string hms(void); }; // Hilfsfunktion: Zeitausgabe

```

gas_station.h

Beispiel: Tankstelle (4)

```
#include "gas_station.h"
extern unsigned errors;
void gas_station::customer1_thread(void) {
  for (;;) { // Simulate gas tank emptying time
    wait((m_full1+rand()%int(m_full1*0.10))*t_MIN);
    // Force 25% of all fill ups to be simultaneous
    // with other customers to check contention
    if (rand()%4==1) wait(e_request2);
    cout << "INFO: " << name() <<
    " Customer1 req. gas (1) at " << hms() << endl;
    m_tank1 = 0;
    // Request fill up, then wait for acknowledge
    do { e_request1.notify(); // I need fill up! (2)
      wait(); // static sensit.: Somebody got filled
    } while (m_tank1 == 0); // Was it us? yes
  } // end forever
} // end customer1_thread()
```

gas_station.cpp

```
void
gas_station::customer2_thread(void) {
  for (;;) { // Simulate emptying time
    wait((m_full2+rand()%
    int(m_full2*0.10))*t_MIN);
    cout << "INFO: " << name() << "
    Customer2 needs gas (1) at " <<
    hms() << endl;
    m_tank2 = 0;
    do { e_request2.notify(); // fillup! (2)
      wait(e_filled); // dynamic sensitivity
    } while (m_tank2 == 0);
  } //endforever
} //end customer2_thread()
```

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


```

void gas_station::attendant_method(void) {
  // ASSERTION: We got here due to either (A) a request in progress
  // (B) an event request from a new customer. Since this is an SC_METHOD,
  // we maintain a small amount of state, m_filling. Initially, we're not filling.
  // Once we get a fillup request, we choose who, initiate filling, and then
  // use dynamic sensitivity to delay by the amount of time it takes to fill the
  // indicated gas tank.
  if (!m_filling) {
    // Check customer 1 first (preferential selection)
    if (m_tank1 == 0 && m_count1 < m_maxcount) {
      cout << "INFO: " << name()
        << " Filling tank1 (3) at "
        << hms() << endl;
      next_trigger(m_filltime*m_full1*t_MIN);
      m_filling = true;

      // Check customer 2 only if no customer 1
    } else if (m_tank2 == 0 && m_count2 < m_maxcount) {
      cout << "INFO: " << name() << " Filling tank2 (3) at " << hms() << endl;

      next_trigger(m_filltime*m_full2*t_MIN);
      m_filling = true;
    } //endif
  } else {

```

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

```

} else {
    // We reach here by timing out on filling the tank, so first update
    // the tank, counts and issue messages about this event for the
    // appropriate customer. Then notify everyone of the event (4)
    if (m_tank1 == 0 && m_count1 < m_maxcount) {
        m_tank1 = m_full1; m_count1++;
        cout << "INFO: " << name() << " Filled tank1 (4) at " << hms() << endl;
    } else if (m_tank2 == 0 && m_count2 < m_maxcount) {
        m_tank2 = m_full2;
        m_count2++;
        cout << "INFO: " << name() << " Filled tank2 (4) at " << hms() << endl;
    } // endif
    e_filled.notify(SC_ZERO_TIME); // We finished filling (4) & are available!
    m_filling = false; // go back to waiting
    // See if we need to stop the simulation
    if (m_count1 == m_maxcount && m_count2 == m_maxcount) {
        cout << "WARN: " << name() << " No more fuel at " << hms() << endl;
        sc_stop();
    } // endif
} // endif
} // end attendant_method()

```

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

Beispiel: Tankstelle (7)

```
#include <sstream>
std::string gas_station::hms(void) {
    std::ostringstream now;
    double mins(sc_simulation_time());
    unsigned days = int(mins/(24*60));
    mins -= days*24.0*60.0;
    unsigned hrs = int(mins/60);
    mins -= hrs*60.0;
    if (days)    now << days << " days ";
    if (days||hrs) now << hrs << " hrs ";
                now << mins << " mins";
    return now.str();
} // end hms()
```

**Hilfsfunktion:
Zeitanzeige**

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

Probleme mit Events

- Events ermöglichen die Simulation von Nebenläufigkeit, erfordern aber sorgfältige Programmierung
 - Events können verpasst werden
 - Handshake-Variable sollte verwendet werden
 - Zeigt an, wenn ein Request anliegt
 - Löschen, wenn Request bestätigt wurde
 - Sicherer Datenaustausch zwischen nebenläufigen Simulationsprozessen

Tankstelle: Channels

- SystemC-Mechanismen erleichtern diese Aufgaben
 - Unterstützung bei der Kommunikation
 - Verkapselung komplexer Kommunikation
- Zwei Arten von Kanälen (Channels)
 - primitive und hierarchische
 - Erstmal: primitive
- Primitive Channels heißen “primitiv”, weil sie weder Hierarchie noch Simulationsprozesse besitzen und daher sehr schnell sind
 - Erben von der Basisklasse **sc_prim_channel**
 - Erben und implementieren auch eine oder mehrere SystemC Interfaceklassen
- SystemC besitzt mehrere eingebaute primitive Channels
 - **sc_mutex**, **sc_semaphore** und **sc_fifo<T>**.

Channels: sc_mutex

- Mutex
 - kurz für “mutual exclusion” – “gegenseitiger Ausschluß”
- Programobjekt, über das mehrere Threads eine gemeinsame Ressource ohne Kollisionen verwenden können
- Während Elaboration wird Mutex mit eindeutigem Namen erzeugt
 - Jeder Prozess, der Ressource benötigt, muss erst den Mutex sperren (lock), damit andere Prozesse diese Ressource nicht verwenden können
 - Nach Benutzung Mutex wieder freigeben
- Andere Threads, die versuchen, Ressource (und Mutex) zu verwenden, müssen warten, bis der aktuelle Besitzer des Mutex diese freigibt (unlock)

Channels: `sc_mutex`

- SystemC realisiert Mutexe mit dem `sc_mutex`-Channel
- Klasse `sc_mutex` implementiert die Interfaceklasse `sc_mutex_if`
 - Verschiedene Zugriffsmethoden, z.B. blockierend und nicht blockierend
 - Blockierende Methoden nur in `SC_THREADS` verwendbar

```
sc_mutex NAME;  
  
NAME.lock(); // Lock the mutex,  
// wait until unlocked if in use  
int NAME.trylock() // Non-blocking, returns success  
  
NAME.unlock(); // Free a previously locked mutex
```

sc_mutex-Beispiel: Tankstelle

- Tankwart ist geteilte Ressource
 - Nur eine Tanksäule → ein Auto gleichzeitig betankbar
- Beispiel: Steuerung eines Autos
 - Nur eine Person gleichzeitig auf Fahrersitz

```
class car : public sc_module {
    sc_mutex drivers_seat;
public:
    void drive_thread(void);
    ...
};

void car::drive_thread(void) {
    drivers_seat.lock(); // sim driver acquires seat
    start();
    ... // operate vehicle
    stop();
    drivers_seat.unlock(); // sim driver leaves
                        // vehicle
    ...
}
```


sc_mutex-Beispiel: Synthese

- In elektronischen Systemen kann **sc_mutex** verwendet werden, um die Zugriffskontrolle (Arbitrierung) für einen gemeinsamen Bus zu modellieren
 - Mehrere Master müssen auf Bus zugreifen können
 - Ohne Entwurf einer Arbitrierung, kann in einem einfachen Modell **sc_mutex** zur Kontrolle verwendet werden
 - Entwurf lässt sich später detaillieren

```
class bus : public sc_module {
    sc_mutex bus_access;
    ...
    void write(int addr, int data) {
        bus_access.lock();
        // perform write
        bus_access.unlock();
    }
    ...
};
```

sc_mutex-Beispiel: Synthese (2)

- Ein Mutex kann auch direkt Teil der Klasse sein, die das Busmodell implementiert
- In einem **SC_METHOD**-Prozess könnte der Buszugriff dann wie folgt aussehen:

```
void grab_bus_method() {  
    if (bus_access.trylock() == 0) {  
        // access bus  
        ...  
        bus_access.unlock();  
    }  
}
```

sc_mutex-Beispiel: Synthese (3)

- **Nachteil:**
 - Freigabe eines **sc_mutex** wird nicht durch Event angezeigt
- Wiederholter Aufruf von **trylock()** notwendig
 - Abhängig von anderem Event oder Zeitverzögerung
- Wenn ein Prozess P1 eine Ressource mit **sc_mutex**, reserviert, muss ein zweiter Prozess, der auf die selbe Ressource zugreifen möchte, **trylock()** in Verbindung mit **wait()** oder **return** aufrufen
 - **wait()** oder **return** zwischen **trylock()**-Aufrufen erforderlich!
 - Sonst ist die Simulation blockiert und P1 kann nicht weiterlaufen, um Ressource freizugeben!
 - Die Simulation würde unendlich oft **trylock()** aufrufen

sc_semaphore

- Modellierung von mehreren Instanzen oder Besitzern für einige Ressourcen erforderlich
 - Beispiel: Parkpositionen auf einem Parkplatz
- SystemC besitzt die **sc_semaphore**-Klasse
 - Erbt von und implementiert die **sc_semaphore_if** Klasse
- Bei Erzeugung eines **sc_semaphore** Objekt muss die maximale Anzahl angegeben werden
 - Ein Mutex ist eine Semaphore mit Anzahl 1

sc_semaphore (2)

- Zugriff auf **sc_semaphore**:
 - Warten auf Verfügbarkeit einer Ressource
 - Benutzen der Ressource
 - Benachrichtigen, wenn Verwendung der Ressource beendet (Freigabe)

```
sc_semaphore NAME (COUNT);

NAME.wait();           // Lock one semaphore
                       // Wait until available if in use
int NAME.trywait()    // Non-blocking, return success

int NAME.get_value() // Returns available semaphores

NAME.post();          // Free one previously locked
                       // semaphore
```

sc_semaphore (3)

- **sc_semaphore::wait()** ist sehr unterschiedlich zur **wait()**-Method bei **SC_THREAD!**
 - Intern wird **sc_semaphore::wait()** implementiert als **wait(event)**
- Tankstelle mit mehreren Zapfsäulen, Selbstbedienung
 - Zapfsäulen als Semaphor modelliert
 - Anzahl = Anzahl verfügbarer Zapfsäulen

```
SC_MODULE(gas_station) {
    sc_semaphore pump(12);
    void customer1_thread {
        for(;;) {
            // wait till tank empty
            ...
            // find an available gas pump
            pump.wait();
            // fill tank & pay
        }
    };
};
```

sc_semaphore: Synthese

- Multiport-Speichermodell mit **sc_semaphore**
 - Semaphore beschreibt Anzahl gleichzeitig realisierbarer Schreib-/Lesezugriffe

```
class multiport_RAM {
    sc_semaphore read_ports(3);
    sc_semaphore write_ports(2);
    ...
    void read(int addr, int& data) {
        read_ports.wait();
        // perform read
        read_ports.post();
    }
    void write(int addr, const int& data) {
        write_ports.wait();
        // perform write
        write_ports.post();
    }
    ...
}; //endclass
```

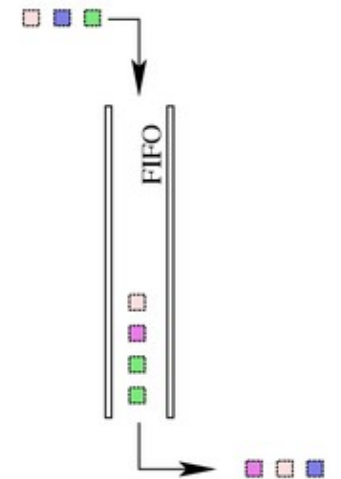
sc_semaphore: Synthese (2)

- Weitere Beispiele
 - Zuteilung von Zeitschlitz in TDM (time division multiplex)-Verfahren, z.B. bei Telekommunikation
 - Kontrolle von Tokens in Token Ring-Netzwerken
 - Schaltinformation für besseres power management

sc_fifo

- **sc_fifo<T>** ist verbreitetster Channel für Modellierung auf Architekturebene
 - First-in first-out Warteschlangen (i.e., FIFOs) werden häufig zur Verwaltung von Datenflüssen verwendet
 - In frühen Phasen des Architekturentwurfs einfach durch die unbeschränkte STL **list<T>** (*einfach verkettete Liste*) implementierbar
- In späteren Stufen des Entwurfs:
 - FIFO-Größen sind bestimmt/bekannt
 - SystemC-Typ **sc_fifo<T>** kann dann detailliertere Modelle beschreiben

First-in First-out (FIFO)



sc_fifo (2)

- Die Klasse **sc_fifo<T>** erbt von und implementiert zwei Schnittstellenklassen
 - **sc_fifo_in_if<T>** und
 - **sc_fifo_out_if<T>**
- Benennung nicht intuitiv!
 - “**in**”-Interface verwendet zum **Lesen** von FIFO und
 - “**out**”-Interface zum **schreiben** in die FIFO
- Die Standardtiefe von **sc_fifo<T>** beträgt 16.
- Datentyp der FIFO-Elemente (**Typname**) muss angegeben werden
 - Eine **sc_fifo<T>** kann beliebige Datentypen enthalten, auch komplexe und große Strukturen
 - z.B. ein TCP/IP-Paket oder einen Festplattenblock

sc_fifo (3)

■ Beispiele:

- FIFOs können Daten zwischen Bildverarbeitungsprozessor und Bus puffern
- Pufferung von Datenpaketen in Kommunikationssystemen während der Übertragung über Netzwerk

```
sc_fifo<ELEMENT_TYPENAME> NAME(SIZE);

NAME.write(VALUE);
NAME.read(REFERENCE);
... = NAME.read() /* function style */
if (NAME.nb_read(REFERENCE)) { // Non-blocking
                               // true if success
    ...
}
if (NAME.num_available() == 0)
    wait(NAME.data_written_event());
if (NAME.num_free() == 0)
    next_trigger(NAME.data_read_event());
```

sc_fifo und KPNs

- Einige Architekturmodelle basieren auf Kahn-Prozessnetzwerken
 - KPNs verwenden *unbegrenzte* FIFOs als Verbindungsmechanismus
- **sc_fifo<T>**
 - *nicht unbegrenzt, blockierendes Lesen/Schreiben*
 - Für KPNs verwendbar, wenn maximale Tiefe bekannt oder bestimmbar ist
 - Bestimmung der Tiefen, so dass Erzeuger und Verbraucher von Tokens keinen Deadlock erzeugen

sc_fifo und KPNs: Beispiel

```
SC_MODULE(kahn_ex) {
    ...
    sc_fifo<double> a, b, y;
    ...
};
// Constructor
kahn_ex::kahn_ex() : a(24), b(24), y(48)
{
    ...
}
void kahn_ex::stim_thread() {
    for (int i=0; i!=1024; ++i) {
        a.write(double(rand())/1000);
        b.write(double(rand())/1000);
    }
}
void kahn_ex::addsub_thread() {
    while(true) {
        y.write(kA*a.read() + kB*b.read());
        y.write(kA*a.read() - kB*b.read());
    } //endforever
}
void kahn_ex::monitor_method() {
    cout << y.read() << endl;
}
}
```

sc_fifo für Software

- Vielfältige Verwendung von FIFOs in Software
 - z.B. als Mailboxen und Warteschlangen
- Bei großen Objekten ist es effizienter, Pointer auf die Objekte auszutauschen
 - Erspart Kopieraufwand
- Bei Verwendung von Pointern sichere Pointer verwenden:
 - z.B. **shared_ptr<T>** aus BOOST
- Für Software-FIFOs ist STL evtl. geeigneter
 - z.B. STL **list<T>**, um unbekannte Anzahl Stimuli einer Testbench zu verwalten
- Prinzipiell liesse sich **sc_fifo<T>** auf Verhaltensebene synthetisieren
 - Abhängig vom Hersteller des Synthesewerkzeugs

Deterministische Modelle synchroner Hardware

- **sc_signal** (und **sc_buffer**) verwenden das *evaluation-update* Paradigma:
 - Jeder Channel hat 2 Speicherstellen:
 - Aktueller Wert und
 - Neuer Wert
 - Writes aktualisieren den neuen Wert
 - Reads lesen den aktuellen Wert
- **request_update()**: von **write()** aufgerufen, als Folge ruft Simulationskern in der Updatephase für jeden Channel, der ein Update anfordert, **update()** auf
- **update()** kopiert den neuen Wert in den alten Wert, kann aber auch Konflikte auflösen oder Events erzeugen
- Evaluate-update-Zyklus ermöglicht *deterministische* Kommunikation

sc_signal

▪ Syntax:

- **sc_signal** <datatype> *signame* [,*signame*] ..;
- *signame*.**write**(*newvalue*)

write beinhaltet Phase, in der Verhalten evaluiert wird und Aufruf von protected **sc_prim_channel::request_update()**;

Aufruf von **sc_signal::update()** ist nicht sichtbar, erfolgt in Updatephase als Folge von **request_update**;

In jedem δ -Zyklus kann nur *ein einzelner Prozess* auf Channels vom Typ **sc_signal** schreiben. Der letzte Wert bleibt gültig.

- *signame*.**read**(*varname*)

sc_signal: Beispiel

```
int                c;
sc_signal<sc_string> sig;
// Initialisierung während 1. Deltazyklus
sig.write("Hello");
c=1;
cout << "c: " << c << " "
      << "sig:" << sig << endl;
wait(SC_ZERO_TIME);
// Zweiter Deltazyklus
c=2;
sig.write("World");
cout << "c: " << c << " "
      << "sig:" << sig << endl;
wait(SC_ZERO_TIME);
// Dritter Deltazyklus...
```

Ausgabe:

c: 1 sig: ''

c: 2 sig: 'Hello'

Tip: Suffix “_sig” für Signale verwenden, um verzögerte Updates anzuzeigen

Gefährlich: Überladener “=”-Operator

- Die Schreibweisen
 - *varname* = *signame*.**read**();
 - *signame* = *newvalue*;
 - *varname* = *signame*;sind auch zulässig
- **Gefährlich**, da bei dieser Schreibweise der evaluate-update Zyklus nicht sichtbar ist!
- Deprecated!



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

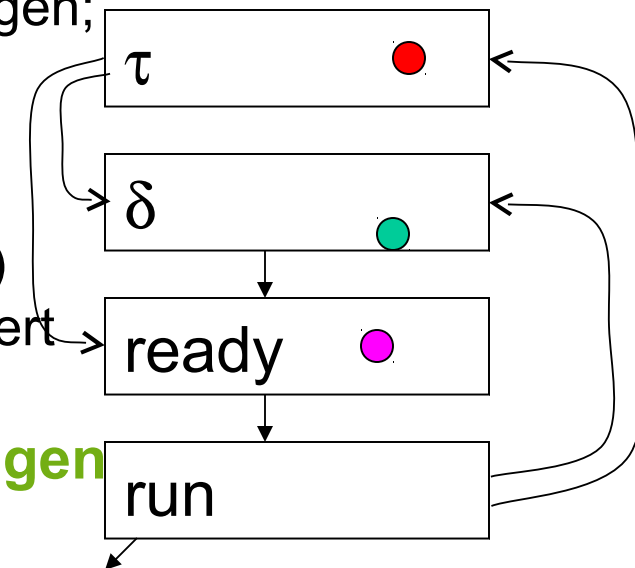
- Sofort (\rightarrow in “ready” stellen),
- Verzögert (\rightarrow in “ δ ” stellen mit Zeit 0)
- Zeitabhängig (\rightarrow Wartezustand mit Zeit)

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

// Makroskopische Zeit

// δ -Zeit

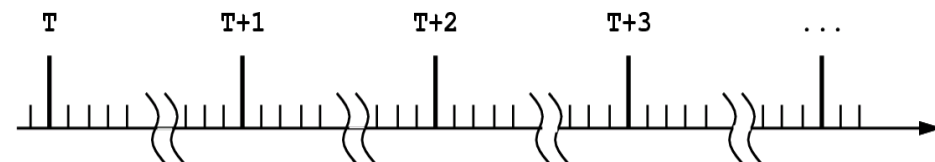
// selbe δ -Zeit



Aktualisiere ausstehende Signaländerungen

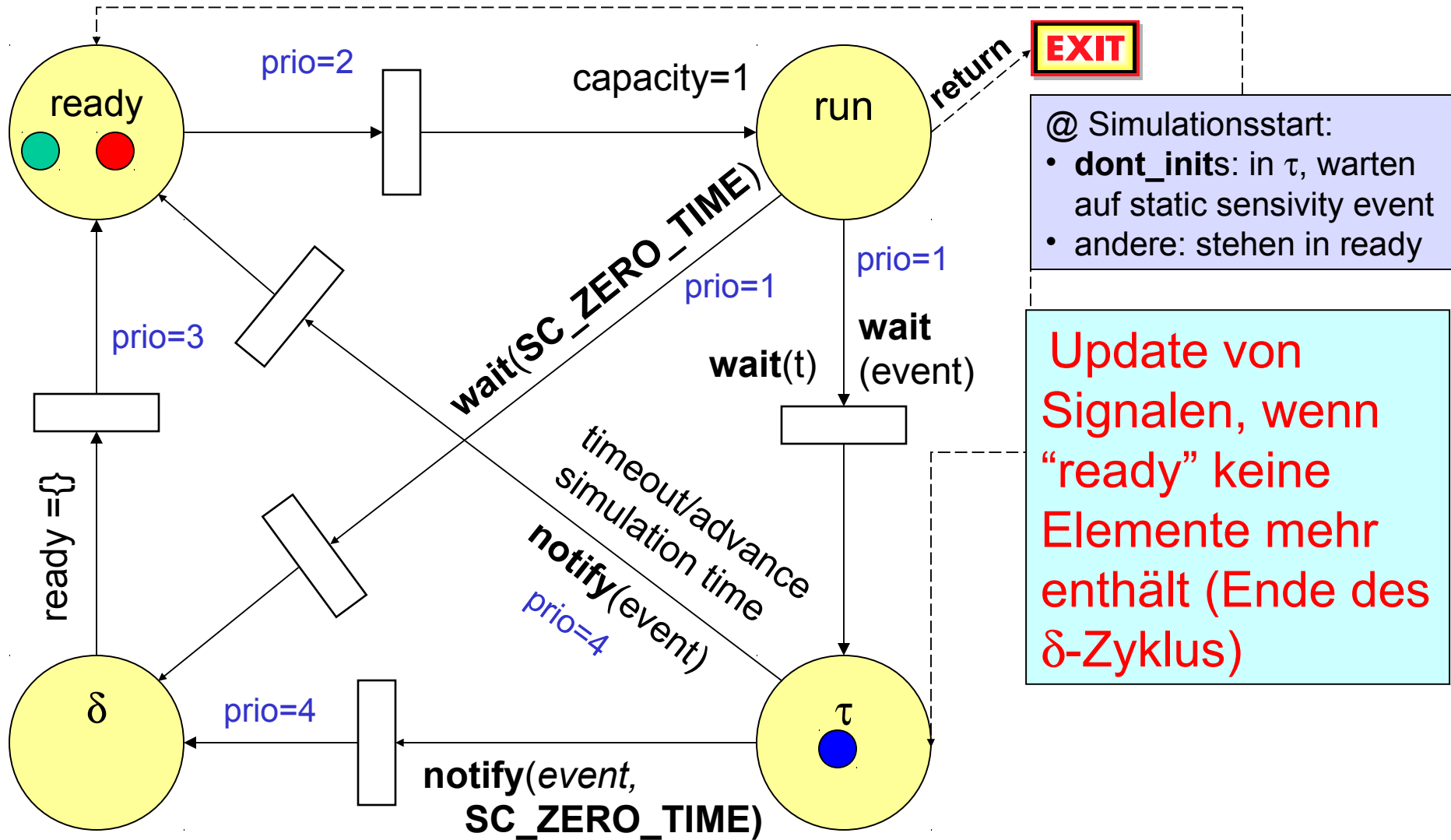
Prozesse in “ δ ” \rightarrow “ready”;

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



Übergänge zwischen Thread-Zuständen

– Prädikats-/Transitionsnetz \approx activity chart \rightarrow Warteschlangenmodell –

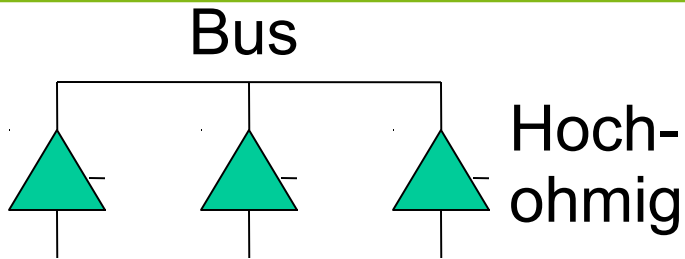


@ Simulationsstart:

- `dont_inits`: in τ , warten auf static sensivity event
- andere: stehen in ready

Update von Signalen, wenn "ready" keine Elemente mehr enthält (Ende des δ -Zyklus)

Mehrfaches Schreiben in einem δ -Zyklus

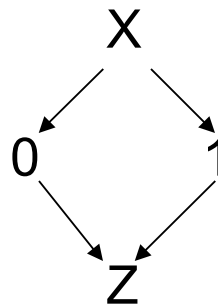


Syntax:

- `sc_signal_resolved name;`
- `sc_signal_rv<width> name; //*`

Semantik identisch zu `sc_signal<sc_logic>`, aber mehrere Schreibzugriffe in einem δ -Zyklus erlaubt. Auflösung ist standardmäßig wie folgt definiert:

| A \ B | '0' | '1' | 'X' | 'Z' |
|-------|-----|-----|-----|-----|
| '0' | '0' | 'X' | 'X' | '0' |
| '1' | 'X' | '1' | 'X' | '1' |
| 'X' | 'X' | 'X' | 'X' | 'X' |
| 'Z' | '0' | '1' | 'X' | 'Z' |



Änderung der Auflösungs-funktion ist unhandlich (siehe Black & Donovan)

* **rv** steht für “resolved vector”

Zusammenfassung

- Tankstellenbeispiel
- Channels
 - **sc_mutex,**
 - **sc_semaphore**
 - **sc_fifo,**
- **sc_signal,**
 - Determinismus
 - ∇ δ -Zyklen
 - **sc_signal_resolved**