technische universität
dortmund

fakultät für informatik
informatik 12

Synthese Eingebetteter Systeme

Wintersemester 2012/13

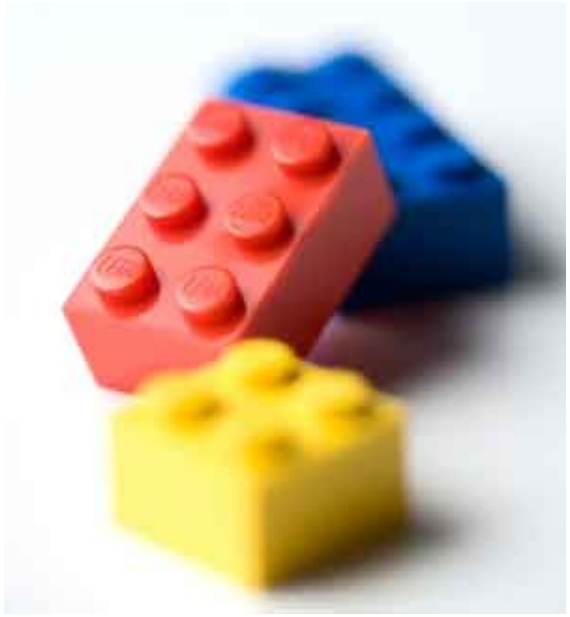
7 – SystemC-Details und Testbenches

Michael Engel
Informatik 12
TU Dortmund

2012/11/12

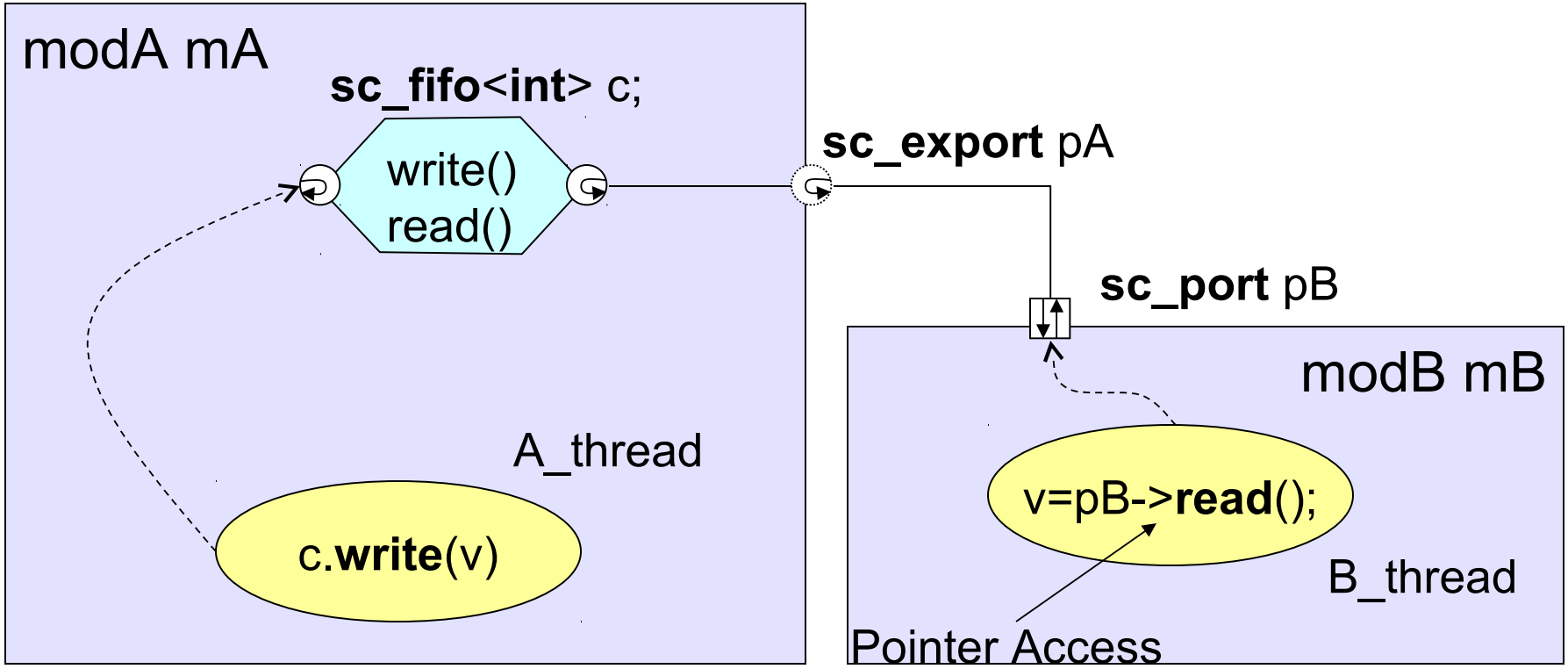
SystemC-Details

- sc_export
- Clocks



sc_export (ab SystemC 2.1)

- Neue Art von Port
  - Ähnlich Standard-Port, Channel innerhalb v. Modulen
  - Ermöglicht einfachere Verbindungen:



Syntax von sc_export

- **Syntax:**
  sc_export<*interface*> *portname*;

Channel-Verbindungen jetzt innerhalb des Moduls:

```
SC_MODULE(modulename) {
    sc_export<interface> portname;
    channel c_instance;
    SC_CTOR(modulename) {
      portname(c_instance);
    }
};
```

Beispiel: Ausgabe umschalten

```
SC_MODULE(clock_gen) {
    sc_export<sc_signal<bool> > clock_xp;
    sc_signal<bool> oscillator;
    SC_CTOR(clock_gen) {
      SC_METHOD(clock_method);
      clock_xp(oscillator); // Verbinde sc_signal channel mit export clock_xp
      oscillator.write(false);
    }
    void clock_method() {
       oscillator.write(!oscillator.read());
       next_trigger(10,SC_NS);
    }};
```

```
#include "clock_gen.h"
clock_gen clock_gen_i;
collision_detector collision_detector_i;
// Verbinde clock
collision_detector_i.clock(clock_gen_i.clock_xp);
```

Einschränkungen

- **sc_export** kann nicht in einer statischen Sensitivitätsliste verwendet werden
  - Folgendes ist aber zulässig: **wait**(*xportname*->*event*()).
- Es existieren keine Arrays von Verbindungen wie bei **sc_port**.

Clocks

- SystemC besitzt den speziellen Channel **sc_clock**
  - Implementiert das **sc_signal_in_if<>**-Interface.
- Clocks erzeugen Timing-Signale
  - Verwendet, um Events zu synchronisieren
- Mehrere Clocks mit beliebigen Phasen sind möglich
- **Syntax:**
  - **sc_clock** *clk*("*clk*", *period*, *duty*, *start*, *pos_first*);
- mit:
  - **sc_time**/**double** *period*: Δ zwischen zwei aufeinander folgenden Taktflanken in gleicher Richtung,
  - **double** *duty*=0.5: % der Periode, in der Clock “True” ist
  - **sc_time**/**double** *start* = 0: Zeitpunkt der ersten Flanke
  - **boolean** pos_first=**true:** ist erste Flanke positiv?

Standardwerte

Clocks: Beispiele

false);

```
sc_time t(10,SC_NS), t2 (5, SC_NS);
sc_clock clk1("clk1", t, 0.5, t2); // 10 ns Periode, 50% duty, Start @ 5ns.
```

```
sc_clock clk2("clk2"); // Periode=1, 50% duty, start @ 0ns; 1st Flanke pos.
```

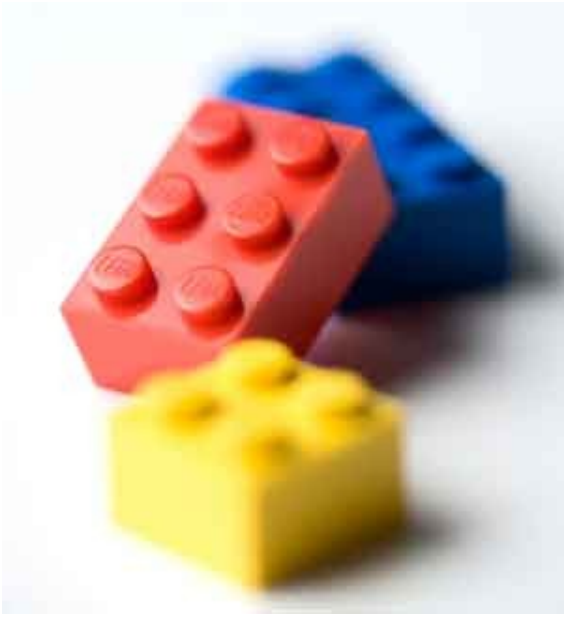
```
sc_clock clk3("clk3",20, 0.1,0, false); // Periode?20, 10% duty, Start@0ns
```

Export von Clocks aus Modulen

```
SC_MODULE(CLOCK_GEN) {
      sc_port<sc_signal_out_if<bool> >  clk_p1;
      sc_export<sc_signal_in_if<bool> > clk_p2; // SystemC 2.1
      sc_clock clk1;
      sc_clock clk2;
      SC_CTOR(clock_gen)
       : clk1("clk1",4,SC_NS), clk2("clk2",6,SC_NS)
      {
        SC_METHOD(clk1_method);
          sensitive << clk1;
          clk_p2(clk2);                                  // schneller
      }
      void clk1_method() {
         clk_p1->write(clk1);
      }
};
```

SystemC-Testbenches

- Dynamische Prozesse
- Debugging und Signalverfolgung
- Testbenches



Dynamische Prozesse (ab SystemC 2.1)

- Funktionen und Methoden können als dynamische Prozesse verwendet werden

Voraussetzung:
#define SC_INCLUDE_DYNAMIC_PROCESSES

Beispiele für als Prozesse verwendbare Funktionen:

```
void inject(); // normale Fkt. ohne Argumente/Rückgabewert
int count_changes(sc_signal<int>& sig) // normale Fkt.
bool TestChan::Track(sc_signal<packet>& pkt); // Methode
bool TestChan::Errors(int maxwarn, int maxerr); // Methode
```

Syntax zur Erzeugung dynamischer Prozesse

- Registrierung dynamischer Prozesse ohne Rückgabewert
- **sc_process_handle** *hname* = **sc_spawn** (
  /* void*/ **sc_bind**(&*funcName*, *ARGS*),
  processName,
  SpawnOptions
  );
- **sc_process_handle** *hname* = **sc_spawn**(
  /*void*/ **sc_bind**(&*methName*, *object*, *ARGS*)
  processName,
  SpawnOptions
  );

Standard:
call by value

Referenz auf
aufrufendes Modul,
z.B. **this**

Stackgröße,
don't_initialize,
sensivity

Debugging und Signalverläufe

- SystemC hat keine Anzeige für Signalverläufe
- Erzeugung von Standard-VCD-Dateien
  - VCD = “value change dump”
- SystemC unterstützt auch proprietäre Formate

sc_tracefile* *tracefile*;
tracefile = **sc_create_vcd_trace_file**(*tracefile_name*);
if (!*tracefile*) **cout** << "There was an error" << **endl**;
…
sc_trace(*tracefile*, *signal_name*, "*signal_name*");
…
sc_start();
…
sc_close_vcd_trace_file(*tracefile*);

Debugging und Signalverläufe (2)

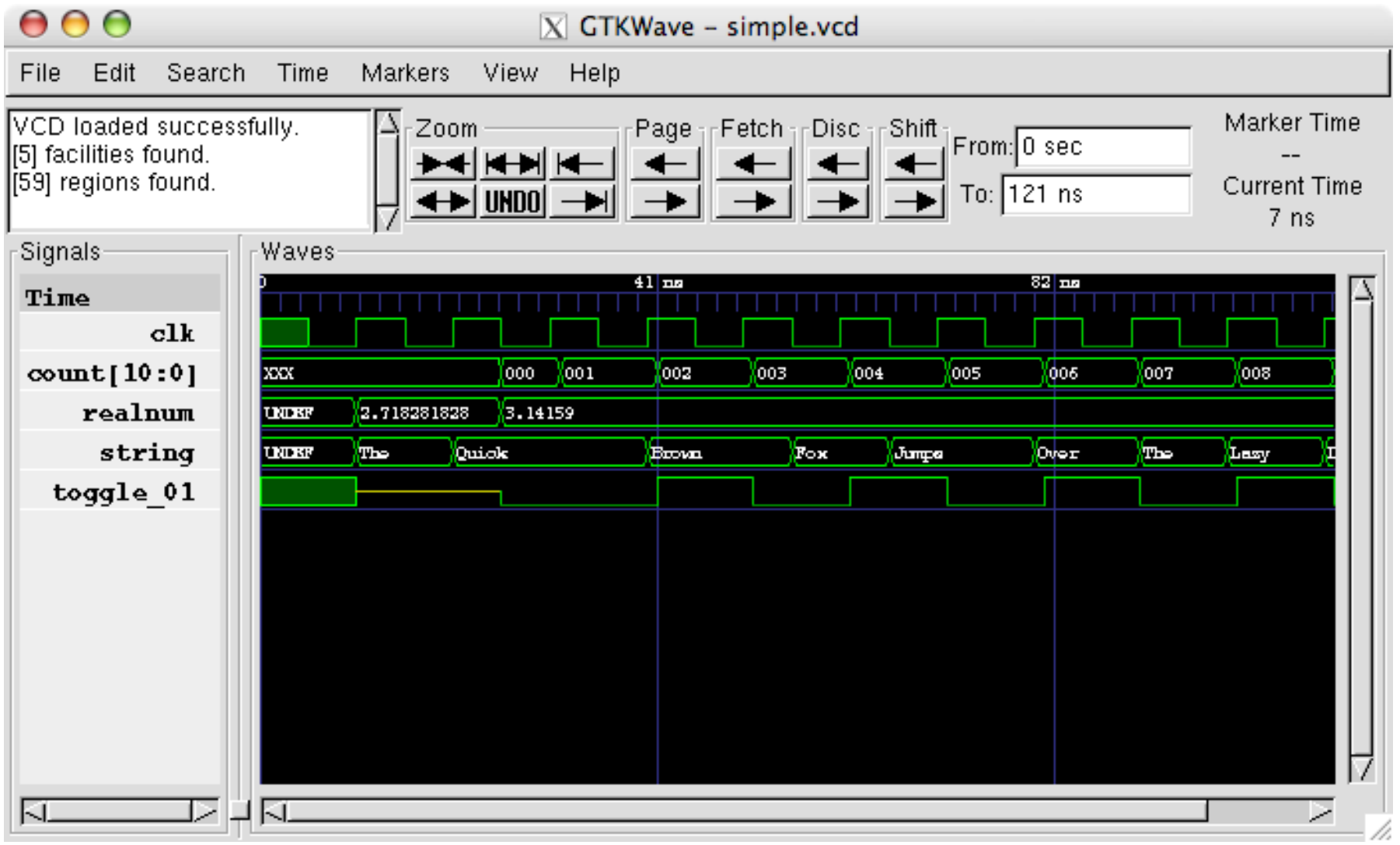
- Signalname muss vor Aufruf von **sc_trace** deklariert werden
- Variablen, Signale und Ports können verfolgt werden
- Dateierweiterung (z.B. “.vcd”) muss explizit angegeben werden
- Tracing kann in Modulkonstruktoren definiert werden

Debugging und Signalverläufe: Beispiel

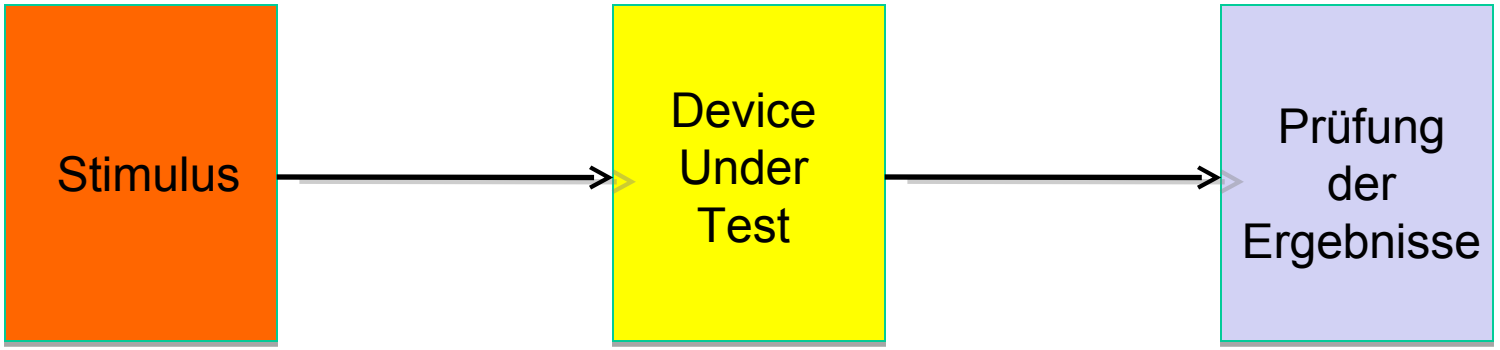
```
// Datei wave.h
    SC_MODULE(wave) {
      sc_signal<bool> brake;
      sc_trace_file* tracefile
      double temperature;
    };
```

```
// Datei wave.cpp
wave::wave(sc_module_name nm);// Konstruktor
:module(nm) { …
  tracefile = sc_create_vcd_file("wave");
  sc_trace(tracefile, brake,"brake");
  sc_trace(tracefile,temperature,"temperature");
} // Ende Konstructor
wave::~wave() {
  sc_close_vcd_file(tracefile);
  cout << “wave.vcd erzeugt" << endl;
}
```

Beispielausgabe: GTKwave



Typische Testbench



- Das Stimulus-Modul erzeugt *Stimuli* für den zu testenden Systementwurf (*Design under Test,* DUT)
- Die Ergebnisprüfung kontrolliert die Ausgabe des Sstems und überprüft, ob die Ausgabe korrekt ist

Testbenches

- Testbenches erzeugen Stimuli für den zu testenden Entwurf und prüfen die erzeugten Ausgaben
- Eine Testbench kann kann auf verschiedene Weisen implementiert werden
- Der Stimulus kann…
  - von einem Prozess erzeugt, Ergebnisse von einem anderen geprüft werden
  - im Hauptprogramm integriert werden und die Ergebnisse von anderem Prozess überprüft werden
- Die Überprüfung kann auch…
  - z.B. im Hauptprogramm integriert werden usw.
- Es gibt keine kanonische Methode zum Entwurf
  - Abhängig von der Anwendung

Typische Testbench

- Das Stimulus-Modul kann Stimuli aus einer Datei lesen oder sie als **SC_THREAD** Prozess realisieren
- Entsprechendes gilt für die Überprüfung der Ergebnisse
- Oft werden Stimuli-Erzeugung und Ergebnisüberprüfung in einem Modul vereint
- Ergebnisüberprüfung kann entfallen, wenn die Ausgabe manuell überprüft wird
  - VCD-Datei => GTKwave

Testbench-Beispiel: Endlicher Automat

- Mealy-Automat:
  - Ein sequentielles Element: Statusvektor
  - Zwei kombinatorische Elemente
    - Ausgabelogik
    - Logik für den Folgezustand

Testbench-Beispiel: Endlicher Automat

- Zustands-diagramm für einen Automaten
- Ausgänge a und b

Testbench-Beispiel: Endlicher Automat

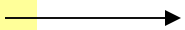
Testbench-Beispiel: Endlicher Automat

```
// main.cpp
```

// Endlos

Testbench-Beispiel: Endlicher Automat

```
// fsm.h
```

Deprecated… 

Testbench-Beispiel: Endlicher Automat

```
// fsm.cpp
```

Testbench-Beispiel: Endlicher Automat

stimulus.h

← Deprecated!

Testbench-Beispiel: Endlicher Automat

stimulus.cpp

Testbench-Beispiel: Endlicher Automat

display.h

← Deprecated!

Testbench-Beispiel: Endlicher Automat

display.cpp

Zusammenfassung

- Exports
- Clocks
- Testbenches