TDTS07: System Design and Methodology
- Laboratory introduction - (Lab 1 and 2)

Adrian Lifa
Embedded Systems Laboratory
Dept. of Computer and Information Science
Linköpings universitet

www.ida.liu.se/~adrli
adrian.alin.lifa@liu.se
Building B, Room 329:206
Where to find me

- Adrian Lifa
  - Office: Building B 329:206
  - Email: adrian.alin.lifa@liu.se
- Organization
- Lab 1
- Break
- Lab 2
Organization

- Lab groups
  - Webreg groups A and B: Adrian Lifa (adrian.alin.lifa@liu.se)
- Web page
  - http://www.ida.liu.se/~TDTS07
  - Check the lab pages for information and links to tutorials
- Organization
  - 2 lessons (including this one)
  - 10 two-hour lab sessions
- 3 lab topics (one tutorial for each)
  1. Modeling and simulation with SystemC
  2. Formal verification with UPPAAL
  3. Design space exploration with MPARM
Lesson 1 – January 22, 15.15-17.00, S10
  - SystemC and timed automata
Lesson 2 – February 12, 15.15-17.00, R27
  - MPARM and UPPAAL demo
Choose a lab partner and sign up for the labs through webreg
  - www.ida.liu.se/webreg (or follow the link from the lab web page)

Deadline for registration: **January 31**
  - do it as soon as possible

Deadline for all lab assignments: **March 22, 2015, 23:59**
  - This is the last day for handing in (emailing) lab reports
  - After the deadline, your assistant will correct the remaining lab reports at his earliest convenience

Rules: Please read them (linked from the lab pages)
1. Modeling and simulation with SystemC
2. Formal verification with UPPAAL
3. Design space exploration with MPARM

- Each lab has a tutorial. Please read it and be prepared before you attend the lab session. We want to work efficiently during the supervised lab sessions.
Examination

- Written report for each lab (pdf)
  - Present your solution to the lab exercises
  - Explain your design and implementation choice in detail
  - Present and discuss your results

- Prepare an archive with the report (pdf) and the code (where any) and email it to your group’s assistant (CC both lab partners)

- Grades:
  - Passed
  - Intermediate: revise your solution and report according to your assistant’s comments
Lab 1 and 2: Modeling, validation, verification

After the system is designed:
- Find out whether the system works according to its specification and correctly
- **Validation** – Quality assurance process
  - "Are you building the right thing?"
- **Simulation**
- Testing
- **Verification** – Quality control process
  - "Are you building it right?"
- **Model checking**
- Theorem proving
Lab 1: SystemC modeling and simulation
Simulation

- Based on executable models of the system
- Generate input stimuli
- Permits a quick and shallow evaluation of the design quality
- Good for finding bugs
- Not suitable for finding subtle errors
- **Compare to VHDL and Verilog**
  - Contains structures for modeling HW components and their interaction
  - Comes with a simulation kernel
  - It’s a unified HW-SW design language

- **What do we need to model systems?**
  - time
  - modules
  - concurrent processes
  - events
  - channels
  - ports
Data type `sc_time` (a C++ class)

Use like an ordinary basic C++ data type (`int`, `double`)

- `sc_time t1(9,SC_MS);`
- `sc_time t2 = sc_time(5,SC_SEC);`
- `if (t1<t2) cout << t1*3 << endl << t2+t2;`

Many of the standard operators are defined for `sc_time`

The underlying representation is based on 64 bits unsigned integer values

- The representable time is limited (discrete time)
- Depends on the time resolution
  - Default: 1 picosecond
  - Can be set by the user through the function `sc_set_time_resolution`
Modules:

- Basic building blocks in SystemC
  - Contains ports, concurrent processes, internal data structures, channels, etc.
- Created with the macro `sc_module`
- Concurrent processes (`sc_thread` or `sc_method`)
  - Use `wait` statements to advance time (or event notification)
  - Sensitive to events (`sc_event`) or value changes in channels
- Input and output ports to communicate with the environment
Example: Adder

![Adder diagram](image)
#include <systemc.h>
#include <iostream>
using std::cout;
using std::endl;

SC_MODULE(Adder) {
  sc_in<int> a_p;
  sc_in<int> b_p;
  sc_out<int> sum_p;
  sc_event print_ev;

  void add_method() {
    sum_p = a_p + b_p;
    print_ev.notify(SC_ZERO_TIME);
  }

  void print_method() {
    cout << sc_time_stamp()
    << ":Sum=\" << sum_p
    << \" endl;
  }

  SC_CTOR(Adder) {
    sum_p.initialize(0);
    SC_METHOD(add_method);
    sensitive << a_p << b_p;
    SC_METHOD(print_method);
    dont_initialize();
    sensitive << print_ev;
  }
}; // END Adder
Generate inputs

Generator

Adder

a

b

sum
SystemC – Test bench

// Definition of an input generator (next slide)

```cpp
int sc_main(int argc, char *argv[]) {
    sc_set_default_time_unit(1,SC_SEC);
    sc_signal<int> a_sig, b_sig, sum_sig; // create channels
    Adder adder_module("Adder_1"); // create an instance
    adder_module(a_sig, b_sig, sum_sig); // connect ports to channels
    Generator gen("Generator_1");
    gen(a_sig, b_sig);
    sc_start(30,SC_SEC);
    return 0;
}
```
SC_MODULE(Generator) {
    sc_out<int> a_p;
    sc_out<int> b_p;
    void gen_thread() {
        for (;;) {
            wait(1,SC_SEC);
            a_p = a_p + 1;
            b_p->write(b_p->read() + 1);
        }
    }
    SC_CTOR(Generator) {
        a_p.initialize(0);
        b_p.initialize(0);
        SC_THREAD(gen_thread);
    }
}; // END Generator
Simulation run

adrli@mina2 ~TDTS30/doc/systemc_lab/examples/adder$ ./adder.x

SystemC 2.1.v1 --- Dec 22 2014 16:12:32
Copyright (c) 1996-2005 by all Contributors
ALL RIGHTS RESERVED

0 s:  Sum=0
1 s:  Sum=2
2 s:  Sum=4
3 s:  Sum=6
4 s:  Sum=8
5 s:  Sum=10
6 s:  Sum=12
7 s:  Sum=14
8 s:  Sum=16
9 s:  Sum=18
10 s: Sum=20
11 s: Sum=22
.
.
.
.
1. **Initialize**: each process executed once; it’s possible to disable this phase for methods

2. **Evaluate**: select a ready to run process and execute/resume it; immediate notification may happen – `e.notify()`

3. repeat 2 until no more processes to run

4. **Update**: values assigned to channels in the previous evaluate cycle

5. step 2-4 = *delta-cycle*; if 2 or 3 resulted in delta event notifications (e.notify(0) or wait(0)) go to 2 **without** advancing simulation time

6. Advance to next simulation time with pending events

Determine processes ready to run and go to 2
Simulator kernel – delta cycle example

//inside a process
sc_signal<int> sig_int;
//assume current value
//of sig_int is 0
sig_int.write(1);
int value = sig_int.read();
cout << value << endl;
wait(SC_ZERO_TIME);
value = sig_int.read();
cout << value << endl;
Try the example

- You can find the adder example at:
  - `/home/TDTS07/tutorial/systemc/adder`
  - Copy it to your home directory

- Two files:
  - `adder.cc` (implements the two modules + the test bench)
  - Makefile (helps you compile and build the program)
    - Type `gmake` at the command line assuming you are in the correct directory
    - Creates an executable `adder.x`

- After building the example, type `./adder.x` to run it

- Study the source code together with the tutorial
Lab assignment

- Study the lab material linked from the course web pages
- At the end of the document you find the lab assignment
  - Design and implement a traffic light controller
Lab 2: Formal verification in UPPAAL
Formal methods

For the levels of complexity typical to embedded systems:

- Traditional validation techniques (e.g., simulation) cover a small fraction of the system behavior
- Bugs found late have a negative impact on time-to-market
- A failure may lead to a catastrophe
Model checking

System Description
Automata model $N$

$\text{AG} \; ! (p_c \& p_d)$
$\text{EF}_{< 2} p_e$

Specification (Req. Properties)

Model Checker

$N \models f$

yes

???

no

Diagnostic Information
Design and verification tool:

- Requirements specification: Computation Tree Logic (CTL)
- Modeling: timed automata
- Validation: simulation
- Verification: model checking

- User-friendly graphical user interface
- www.uppaal.com (free download for academic use)
A timed automaton is a finite automaton augmented with a finite set of real variables called \textit{clocks}.

All the clocks change along the time with the same constant rate.

Timing constraints can be expressed imposing \textit{conditions} over clocks.

The timed automata model consists of a collection of automata that operate and coordinate with each other through \textit{shared variables} and \textit{synchronization labels}.
Timed automata (syntax)

Clocks: X, Y

Guard = clock constraint

X <= 5 & Y > 3

Reset (action performed on clocks)

X = 0

Invariant

Y <= 2

Action used for synchronization

n

m
Timed automata (example)
Temporal logics

How do we specify properties for timed systems?

- Logic augmented with temporal modal operators
  - Allow us to reason about how the truth of assertions changes over time

- Used to specify desired properties of timed systems
  - Safety: Nothing bad will ever happen
  - Liveness: Something good may eventually happen
  - Bounded response: Something will happen within a time limit

- Different forms of temporal logic
  - depending on the underlying model of time
  - we use Computation Tree Logic (CTL)
CTL: Computation Tree Logic

- Based on propositional logic of branching time: it may split into more than one possible future
- Atomic propositions (states in the TA) and boolean connectors
- Temporal operators:
  - Path quantifier
    - A – all computation paths
    - E – some computation path
  - Forward-time operators
    - G – globally
    - F – in the future
    - X – next time
    - U – until
Computation tree

- Represents an unfolded state graph – nodes are the possible states that the system might reach
- Build (infinite) computation trees from the TA model
CTL temporal operators

\( \text{EX } p \)  
\( \text{EF } p \)  
\( \text{EG } p \)  
\( \text{AX } p \)  
\( \text{AF } p \)  
\( \text{AG } p \)

Possible  
Potentially global  
Inevitable  
Invariant
Two properties of the system

- Possible error: \textbf{EF} Error (True!)
- Possibly globally OK: \textbf{EG} OK (True!)
- \textbf{AG} OK (False!!)
UPPAAL has a special syntax for CTL

- AF $p = A<> p$  
  *Always eventually $p$*

- AG $p = A[] p$  
  *Invariantly $p$*

- EF $p = E<> p$  
  *Possibly $p$*

- EG $p = E[] p$  
  *Potentially always $p$*

- Boolean connectives: and, or, not, imply

- No nested formulas, except
  - A[] ($p$ imply A<> $q$) = $p \rightarrow q$
  - $p \rightarrow q$ “If $p$ holds in some state, then $q$ will hold in some future state”

This construction can be used to verify one of the properties in the assignments: “If a car arrives at the traffic light, it should eventually be granted green light.”
Formal methods can

- overcome some of the limitations of traditional techniques,
- give a better understanding of the system,
- help to uncover ambiguities, and
- reveal new insights of the system.

Formal methods do have limitations and are to complement simulation and testing, rather than replacing them.
Lab assignment

- Study the lab material linked from the lab pages
- In the document you will find the lab assignments as well as the requirements on the deliverables
  - Introductory assignments to get familiar with timed automata, CTL, and UPPAAL
  - Model the traffic light controller in timed automata
    - Simulate and verify.
  - Implement a communication protocol
    - Alternating Bit Protocol
- Design space exploration with MPARM
  - HW
  - SW
  - Design space exploration for energy minimization
  - Communication
  - Mapping and scheduling
Thank you!
Questions?